Formally sound implementations of security protocols with JavaSPI

Author's Accepted Manuscript of article published in Formal Aspects of Computing
(published version link: https://link.springer.com/article/10.1007/s00165-017-0449-8)

Formally Sound Implementations of Security Protocols with JavaSPI

Riccardo Sisto[1] and Piergiuseppe Bettassa Copet[1] and Matteo Avalle[1] and Alfredo Pironti[2]

[1]Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy

[2]IOActive, Spain

Abstract. Designing and coding security protocols is an error prone task. Several flaws are found in protocol implementations and specifications every year. Formal methods can alleviate this problem by backing implementations with rigorous proofs about their behavior. However, formally-based development typically requires domain specific knowledge available only to few experts and the development of abstract formal models that are far from real implementations.

This paper presents a Java-based protocol design and implementation framework, where the user can write a security protocol symbolic model in Java, using a well defined subset of the language that corresponds to applied π-calculus. This Java model can be symbolically executed in the Java debugger, formally verified with ProVerif, and further refined to an interoperable Java implementation of the protocol. Soundness theorems are provided to prove that, under some reasonable assumptions, a simulation relation relates the Java refined implementation to the symbolic model verified by ProVerif, so that, for the usual security properties, a property verified by ProVerif on the symbolic model is preserved in the Java refined implementation.

The applicability of the framework is evaluated by developing an extensive case study on the popular SSL protocol.

Keywords: Security protocols; Formal methods; Formal verification; Model-driven development

1. Introduction

Security protocols are network protocols that protect exchanged data against network attackers. Typically, security is achieved by means of cryptography, hence such protocols are also called cryptographic protocols.

Unfortunately, bare use of cryptography alone is not sufficient. The security protocol design may not completely satisfy its intended security properties (e.g. [RRDO10, BLF+14]), or it may incorrectly use cryptographic primitives (e.g. [AFP13, APW09]). Furthermore, implementations may have coding bugs that affect security (e.g. [App14, Gnu14, Hea14]).

Formal support in the design of security protocols and the development of their implementations can

Correspondence and offprint requests to: Riccardo Sisto, Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy. e-mail: riccardo.sisto@polito.it phone: +390110907073

increase the assurance in the final artifact. Formal methods work by setting a *model* that unambiguously defines attacker capabilities, protocol parties' behavior, and expected security properties. Rigorous reasoning can be done on the model about the interaction of the attacker with the protocol parties, assessing whether the security properties hold or not (although, in general, such a problem is known to be undecidable [HT96]).

In practice, security protocol implementations are rather complex artifacts, which are hard to analyze; hence it is often the case that a more abstract model of the protocol is analyzed, instead of a widely deployed implementation. An abstract model captures the essential features of the protocol, ignoring distracting implementation details. Hence, using an abstract model makes the protocol behavior clearer to understand, and allows the developers to focus separately on protocol design aspects first, and low-level development aspects later. However, using an abstract model also creates some issues: on one hand, there is a gap, and hence a potential disconnection, between the analyzed model and the running implementation; on the other hand, the abstract model is typically expressed in some formal language that is far from the typical development languages, thus making the above mentioned gap even larger than necessary. Moreover, formal languages have generally limited tool support and are not widely known.

The JavaSPI framework is presented in this paper to address the above issues in a uniform development environment already familiar to many users. With JavaSPI, a security protocol developer can:

- Write security protocol models using a defined subset of the Java programming language;
- Symbolically execute the Java model using a standard Java debugger;
- Express security properties as standard Java annotations, and automatically derive a ProVerif [Bla09] model for their formal verification;
- Refine the Java model via standard Java annotations, and automatically obtain an interoperable implementation of the security protocol;

The subset of Java used to specify models is defined with the aim of enabling the same expressiveness of the applied π-calculus accepted by ProVerif, but with a Java syntax. On one side this facilitates the establishment of a sound correspondence between the Java and applied π-calculus models, and on the other side it allows the user to write models in Java, albeit with restrictions, but still keeping the possibility to exploit the many productivity tools available for Java.

The two translation steps from a Java model to the corresponding ProVerif model and from a Java model to a refined Java interoperable implementation are formally proved to preserve the protocol semantics, so that the security properties verified on the ProVerif model are preserved down to the interoperable Java implementation. In previous papers [APSP11, APPS11] JavaSPI was introduced in a descriptive way, without a full formal treatment. In this paper we fill this gap by adding (i) a formal definition of the JavaSPI language and of how it is automatically translated into ProVerif and refined into a Java implementation, and (ii) a proof of the soundness result.

The remainder of the paper is organized as follows. Section 2 discusses related work and Section 3 describes the JavaSPI framework and gives the formal definition of JavaSPI models. Then, Section 4 introduces the approach and the steps necessary for proving the soundness of the refinement procedure, and the next sections develop such steps. Section 5 formally defines the JavaSPI-ProVerif translation. Section 6 formally defines the refinement from an abstract JavaSPI model into the Java classes that implement it. Section 7 formulates a semantics for JavaSPI models and their implementations, and Section 8 formulates the Soundness theorem and provides its proof. Finally, Section 9 discusses the security guarantees provided by the framework on the final protocol implementation, Section 10 presents a case study of the SSL handshake protocol, and Section 11 concludes.

2. Related Work

In the literature, mainly three approaches have been documented on the verification of security protocol implementations. (i) Code generation starts from a verified formal abstract model, and generates executable code from it; (ii) model extraction goes in the opposite direction, starting from implementation code, and extracting a formal model which is then verified; finally (iii) refinement type checking starts from both implementation code and a model expressed as types, and checks that the code behaves according to what is specified by the types.

To our knowledge, JavaSPI is the first framework that uses a hybrid approach in which a programming

language is used to define an executable symbolic model, and then both model extraction and code generation are leveraged respectively for formal verification and refinement into an interoperable implementation. Furthermore, with the exception of [BFGT08], no existing work allows for the interactive simulation of a symbolic protocol execution.

In the following, the most relevant work in the three existing branches will be discussed; a thorough analysis of the existing techniques can be found in [APS14].

Cade and Blanchet [CB12] present a code generation framework where an OCaml implementation of a cryptographic protocol can be synthesized starting from a CryptoVerif specification. The paper shows a generated implementation of the SSH protocol that successfully interacts with OpenSSH. In [CB12], protocol verification is performed in the computational model of cryptography, which is more precise than the symbolic model used by JavaSPI. Development of the CryptoVerif model requires knowledge of the domain-specific language, and symbolic model execution is not possible. Furthermore the OCaml target language is less common than Java, which may hinder embedding of the generated implementation into larger security-aware applications.

Kiyomoto et al. [KOT08] propose a tool that can generate C code from an XML document containing the protocol actors' specifications. Neither the XML specification can be verified for security, nor is the XML to C translation proved to preserve the protocol semantics.

Other approaches like [ABB$^+$10, BCD$^+$09, SPP01] synthesize security protocol models from high-level service orchestration specifications where no cryptography is explicitly used. The synthesized security protocol models can be formally verified and implementations can be derived. Such works differ in scope from JavaSPI, in that they design new *ad hoc* security protocols, while JavaSPI aims at obtaining interoperable implementations for existing security protocols.

Almeida et al. [ABBD13] automatically generate assembly code (PowerPC, ARM and x86) from C implementations of cryptographic primitives, like RSA-OAEP. The C code is verified for security in the computational model and for absence of control flow side-channels. The compilation into assembly code is proved to be semantics preserving. The work in [ABBD13] is complementary to JavaSPI, in that the former focuses on the verification of implementations of cryptographic primitives, while the latter assumes them to be secure and focuses on the correctness of security protocol implementations.

Avanesov et al. [ACMR12] translate Alice-and-Bob protocol notations of web services into the ASLan domain specific language, which formally expresses orchestration of web services and their security properties. Since Alice-and-Bob notation does not have formal semantics, the ASLan translation is conservative, in that all possible checks on the received content are implemented. The ASLan model only supports one execution path, which rules out support for conditional protocol messages and error messages. The ASLan model can be compiled into a Java servlet application, but the translation is not proved to preserve the protocol semantics. The generated Java servlet is not interoperable, neither in the message format nor in the negotiation of cryptographic algorithms and parameters.

Among the model extraction approaches, Aizatulin et al. [AGJ12] propose a technique that automatically extracts a CryptoVerif model from a security protocol implemented in C, thus enabling security proofs in the computational model. The model extraction is proven sound, so that verification of the CryptoVerif model implies the security of the original C program. The techniques presented in [AGJ12] work by creating a symbolic run of the protocol, where only one execution path is considered. Hence, this approach cannot directly cope with real-world protocols like TLS, where the choice of different algorithms implies different message exchanges, or where error conditions are handled in conditional branches by sending error messages. Another work based on model extraction for protocols implemented in C is [DGJN14], which exploits a general purpose C verifier and is less restrictive about the C code that can be analyzed.

Another framework based on model extraction is Elyjah [O'S08], which takes a complete Java implementation of a security protocol, and extracts a verifiable formal model. The model extraction algorithm is not proved to be semantics preserving. By comparison, leveraging Java annotations a JavaSPI model clearly separates implementation and verification details from the Java model. Additionally, the JavaSPI model can be symbolically executed and soundness proof for the translation steps is provided.

In [BFGT08], a ProVerif model is extracted from an F# implementation, and the extraction algorithm is proved to preserve the implementation semantics. Furthermore, the F# implementation can be linked against both symbolic and concrete implementations of the cryptographic libraries, allowing a form of symbolic protocol execution similar to the one offered by JavaSPI. In [BFGT08] however, one first has to write a full implementation, including all the cryptographic details such as key lengths and algorithms, before it can be symbolically executed; by contrast, in JavaSPI the Java model need not be refined to be symbolically

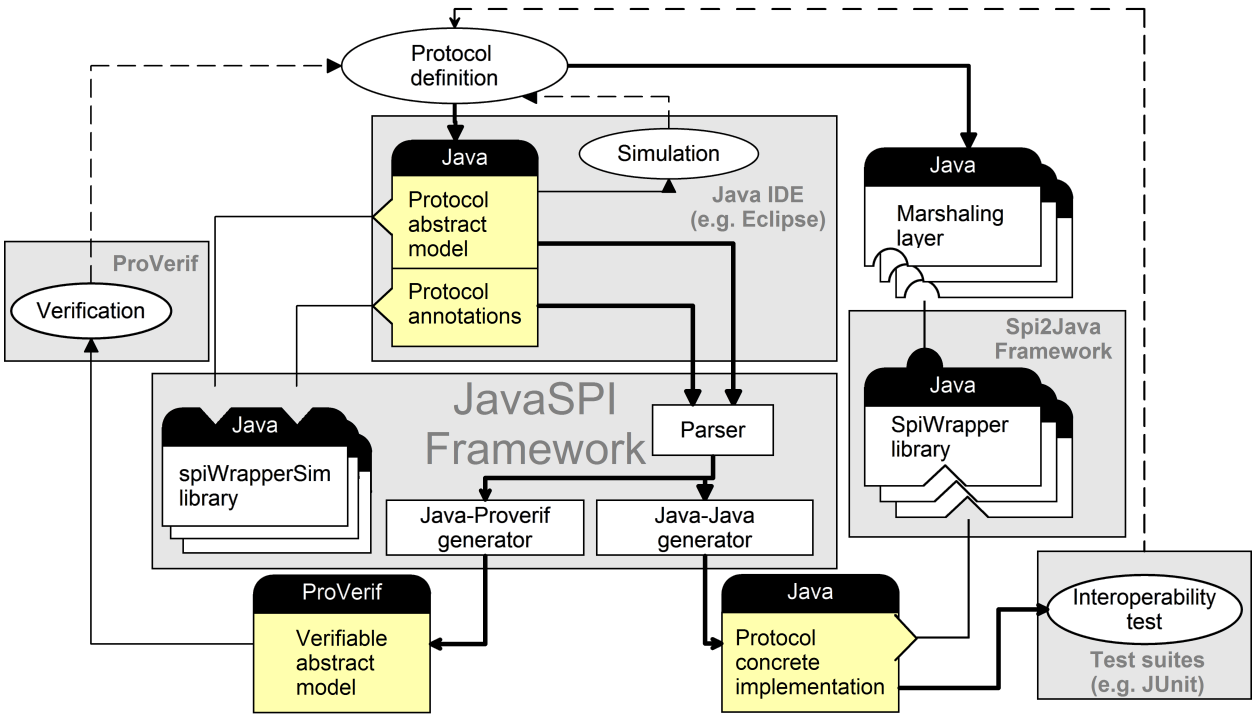

Fig. 1. The complete workflow provided by JavaSPI.

executed, which helps in rapid prototyping and in keeping separate the protocol design aspect from the cryptographic interoperability aspect. Finally, the F# language is not as popular as Java, thus making the learning curve steeper for the typical developer.

Jürjens [Jür05] and Basin et al. [BDL06] extend the UML modeling language by adding security-specific features to enable the representation and verification of security protocols as UML diagrams. Java code generation from these models is also possible [MJH+10], and a refinement approach which starts from these models and preserves secrecy properties has been proposed in [Jür01].

Bettassa Copet et al. [BCPP+12] define a domain-specific visual language for the specification of security protocols. The models written in this language can be translated into ProVerif for formal verification, and from ProVerif into interoperable Java implementations. The visual language lacks formal semantics, so its translation to ProVerif cannot be proved sound. Overall, these visual approaches take a complementary approach to JavaSPI, where developer-friendly formal models are achieved by visual means, as opposed to resorting to well-known implementation languages.

3. The JavaSPI Framework

JavaSPI [APSP11, APPS11] is a framework based on Spi2Java [PS07] that enables formal symbolic modeling and model-driven implementation of cryptographic protocols. The formal symbolic models are based on the Dolev-Yao abstraction [DY83]. Both the formal symbolic model and the implementation of a protocol are written in Java, using a subset of the whole Java language and the JavaSPI libraries.

The workflow of JavaSPI is shown in Figure 1. Initially, the user creates a protocol symbolic model which can be simulated and debugged by running and debugging the Java code of the model itself; this can be done using any of the many productivity tools available for Java.

For each protocol symbolic model written in Java, the Java-ProVerif tool, which is part of JavaSPI, can generate a semantically equivalent model written in applied π-calculus [AF01]. The applied π-calculus is

a formal specification language for security protocols based on the Dolev-Yao abstraction. The generated model can be fed to ProVerif [Bla01], an efficient automated theorem prover for security protocols.

The code of the Java model can be enriched with Java annotations to specify the intended security properties of the protocol. Such annotations are automatically translated by the Java-ProVerif generator into queries for the ProVerif tool. The tool can prove or disprove the intended properties on the model, with the ability to provide counterexamples.

JavaSPI generates ProVerif queries to prove two kinds of security properties, namely secrecy of protocol data and correspondence of events. The latter class includes authentication properties. For example, an authentication requirement could be expressed as $terminate(A, B, x) \Rightarrow start(B, A, x)$, which means that each time actor A terminates a session of the protocol apparently with B agreeing upon some data x (i.e. event $terminate(A, B, x)$ occurs), B has previously started a session of the protocol with A agreeing upon the same data x (i.e. event $start(B, A, x)$ has occurred).

A Java implementation of the symbolic model (i.e. a real, interoperable implementation of the protocol) can be semi-automatically generated from the Java model. This code generation process requires a preliminary manual refinement of the model code with Java annotations that specify some implementation choices (e.g. the algorithms to be used for each encryption operation, or the way message fields have to be serialized). Then, the Java-Java generator performs the generation of the implementation code. This approach lets us have a clear separation between implementation details (annotations) and abstract protocol model (Java statements). Also, it enables a separation of concern in the development of the protocol implementation (first the developer focuses on the abstract model only, then implementation details are added following a sort of aspect-oriented approach). Finally, this approach simplifies the hand-written Java code of the protocol role description because the final, more complex, implementation code is automatically generated.

In the abstract Java model, cryptographic and network operations are executed symbolically, by means of the SpiWrapperSim library, which implements these operations according to the Dolev-Yao modeling style. In the protocol implementation, instead, these operations are executed concretely, by means of the SpiWrapper library, which replaces SpiWrapperSim and implements these operations by delegating them to the underlying Java cryptography and network libraries.

The marshaling and unmarshaling operations, which are responsible for serializing and deserializing messages according to the protocol data formats, constitute the so-called marshaling layer, which is used by SpiWrapper in order to serialize data before sending or encrypting them or for deserializing received or decrypted data. The marshaling layer can be hand-written or, in some cases (when the serialization rules are based on standard data representations such as ASN.1 or XML data types), it can be generated automatically by the Java-Java generator. Specific annotations specify the marshaling layer to be used or the standard rules for its generation. The automatic generation of the marshaling layer and the annotations for specifying security properties are new features that were not present in the framework as described in [APPS11].

The next sections give a formal definition of the Java subset used for specifying the abstract model and describe the annotation system.

3.1. Writing Abstract Models

A security protocol involves two or more *actors* (also called "processes"), each one playing a protocol role. In JavaSPI the behavior of each role is specified by a different Java class. This class must extend the *spiProcess* class included in the SpiWrapperSim library, which also incorporates the code needed for simulation. The class that inherits from *spiProcess* must define the *doRun()* method, which is the abstract description of the protocol role behavior. This method can be defined to take an arbitrary number of arguments. A protocol scenario (i.e. the specification of which protocol roles are instantiated and with what actual arguments) can be specified by means of another class that also inherits from *spiProcess*. The *doRun()* method in this case has no arguments and is just used to instantiate the protocol roles appropriately.

The behavior specified in the *doRun()* method of a protocol role class corresponds to the execution of that role in a single protocol session. Generally, the abstract description of this behavior is rather simple, because it includes one or more possible finite sequences of elementary actions (building and sending messages, receiving and splitting messages, performing cryptographic operations), selected according to simple conditions, such as equality tests or exceptions occurring in the cryptographic or communication operations. This simplicity is reflected in the domain-specific languages for the abstract specification of security protocols, such as for example the applied π-calculus.

```
public class A extends spiProcess {

 public A(final Message...arg) {
  super(arg);
 }

 public void doRun(final Identifier plainMsg, final SharedKey sk, final Channel cAB)
    throws SpiWrapperSimException {

  final Nonce msgNonce = new Nonce();
  final Pair<Identifier,Nonce> msgPair = new Pair<Identifier,Nonce>(plainMsg,msgNonce);

  final Nonce IV = new Nonce();
  final SharedKeyCiphered<Pair<Identifier,Nonce>> Mk = new
  SharedKeyCiphered<Pair<Identifier,Nonce>>(msgPair,sk);
  final Pair<SharedKeyCiphered<Pair<Identifier,Nonce>>,Nonce> p = new
  Pair<SharedKeyCiphered<Pair<Identifier,Nonce>>,Nonce>(Mk,IV); cAB.send(p);

  final Hashing x = cAB.receive(Hashing.class);
  final Hashing Hm = new Hashing(msgPair);
  if (x.equals(Hm)) {
    cAB.send(x);
  } else fail();
 }
}
```

Fig. 2. A sample JavaSPI model of a protocol actor

The subset of Java used in JavaSPI for modeling was defined with the aim of providing the same expressiveness as the applied π-calculus, but with features that would make it amenable to typical Java developers. For this reason, two Java subsets have been defined: a core language, used only internally by the JavaSPI tools, which has been kept very close to the applied π-calculus and as simple as possible, in order to facilitate formal proofs and automatic handling; and an extended language, which is offered to the user and enables several additional features of the Java language.

Forcing the user to use the core language for the specification of protocol roles would be possible, but it would make protocol specification not so immediate and somewhat unnatural for Java programmers who are used to having the whole Java language at their disposal. For this reason, the extended language has been introduced, which improves the user experience in specifying protocols, by allowing a wider set of Java constructs. The additional constructs allowed in the extended language have been properly selected, so as to ensure that they can be automatically translated into the core language, by means of code refactoring operations (i.e. without altering the observable code behavior).

Figure 2 shows a simple JavaSPI model of a protocol role named `A`. This is just an example that is used throughout this paper to show how JavaSPI works. A complete case study of the SSL protocol is described in Section 10. The example process shown in Figure 2 receives three arguments from the caller: an identifier object (i.e. a literal message, named `plainMsg`), a shared key `sk`, and the channel `cAB` used by the protocol for communication. The process initially creates a nonce `msgNonce` and a pair `msgPair` which contains the identifier received from the caller and that nonce. Another nonce, `IV`, is used as initialization vector for encryption. The process encrypts `msgPair` with a shared symmetric key `sk` and sends the encrypted value together with the initialization vector (combined in a pair object `p`) on the channel. On the other side of the channel, another listening process (not listed in Figure 2) decrypts the received value, calculates its hash, and sends it to the original process (`A`). Then, `A` calculates the hash digest `Hm` of the local value `msgPair`, and compares it with the received value `x`. If the two values are equal, the process sends the received value `x` on the channel, otherwise it fails, by calling the `fail()` method. This method terminates the protocol session immediately.

A constructor, which calls the parent constructor, is required in a JavaSPI abstract model class, in order to obtain a model that can be properly simulated.

Figure 3 shows a sample protocol scenario class that uses the role class `A` defined in Figure 2 and another role class `B`. In this case, *doRun()* just creates a number of common data objects (a plain message $plainMsg$,the shared key sk, and a communication channel cAB), specifies how the protocol roles A and

```
public class p_Master extends spiProcess {
        public void doRun() throws SpiWrapperSimException {

                final Identifier plainMsg = new Identifier("A␣security␣critical␣message");
                final Nonce n = new Nonce();
                final SharedKey sk = new SharedKey(n);
                final Channel cAB = new Channel();

                final A npa = new A(plainMsg, sk, cAB);
                final B npb = new B(sk, cAB);

                start(npa,npb);
        }
}
```

Fig. 3. A sample JavaSPI model of a protocol scenario

B have to be instantiated, and finally starts the execution of an unbounded number of A and B instances (by calling *start()*). Note that the *start()* method starts the instances of the protocol roles by calling their *doRun()* methods with the same arguments that have been passed to the role object constructor ($plainMsg$, sk, and cAB for A in the example).

3.2. The Core Language

As the model to be specified is symbolic, the Java data types used for this purpose must be symbolic too. Accordingly, the core language used to specify the *doRun()* method admits, as data types for variables and constants, only the core classes that belong to the SpiWrapperSim library and that represent the classical abstractions used in symbolic models of security protocols (opaque messages, nonces, keys, messages encrypted with different kinds of encryption schemes, ...). Each one of these classes corresponds to one of the data abstractions typically used in applied π-calculus, with its constructor and destructor functions.

For example, shared-key encryption is generally represented in applied π-calculus by means of a pair of functions: a constructor function (SymEncrypt) that takes two arguments (a plaintext and a shared key) and builds the encryption of the plaintext with the shared key, and a destructor function (SymDecrypt) that takes an encrypted message and a shared key and returns the plaintext. This function succeeds only if its first argument is the result of an encryption operation performed with the same key. This model is represented in the syntax accepted by ProVerif as shown in Figure 4a, which means that SymEncrypt is a function with two arguments and SymDecrypt is a function that succeeds only if its arguments take the form SymEncrypt(M, Key) and Key, respectively. In that case, SymDecrypt returns M.

In JavaSPI this data abstraction is represented by means of the Java classes Message, SharedKey, and SharedKeyCiphered. Message represents a generic message (it is the root of the data class hierarchy, shown in Figure 5), hence it can be used for the plaintext. SharedKey represents a key used for shared-key encryption, and SharedKeyCiphered represents the result of a shared key encryption operation. The SharedKeyCiphered class is defined as shown in Figure 4b.[1] Its constructor corresponds to the ProVerif constructor function SymEncrypt while its decrypt method corresponds to the ProVerif destructor function SymDecrypt. In general, ProVerif constructors are represented in JavaSPI by data class constructors while ProVerif destructors are represented in JavaSPI by the other methods available in the data classes. Generally, in JavaSPI, each destructor that can fail is made available in two different forms: a destructor that in case of failure throws an exception and another destructor that lets the process detect the failure and react accordingly. This second kind of destructor (not shown in Figure 2) never throws exceptions but it returns a pair made of a boolean, which specifies whether the operation was successful or not, and the result of the operation, which is set to an invalid value if the boolean is false. For example, the *SharedKeyCiphered* class, in addition to the *decrypt()* method shown in in Figure 2, which returns a *Message* and throws an exception if decryption fails, has a *decrypt_ w()* which returns a *ResultContainer<Message>* object, a commodity entity with a boolean

[1] For simplicity, here only the main public features of the class are shown.

(a) ProVerif model.

```
1 fun SymEncrypt/2.
2 reduc SymDecrypt(SymEncrypt(M, Key), Key) = M.
```

(b) Java class that represents an encrypted message.

```
package it.polito.javaSPI.spiWrapperSim;

public class SharedKeyCiphered<M extends Message> extends Message {

    private final M payload;
    private final SharedKey shKey;

    public SharedKeyCiphered(final M payload, final SharedKey shKey) {
        super();
        this.payload = payload;
        this.shKey = shKey;
    }

    public M decrypt(final SharedKey shKey) throws SpiWrapperSimException {
      if (!this.shKey.equals(shKey)) {
        throw new SpiWrapperSimException("shared key decrypt failed");
      }
      return payload;
    }

    @Override
    public boolean equals(final Object obj) {
      //compare type, payload and shKey
      ...
    }
}
```

Fig. 4. Shared key encryption model in ProVerif and JavaSPI.

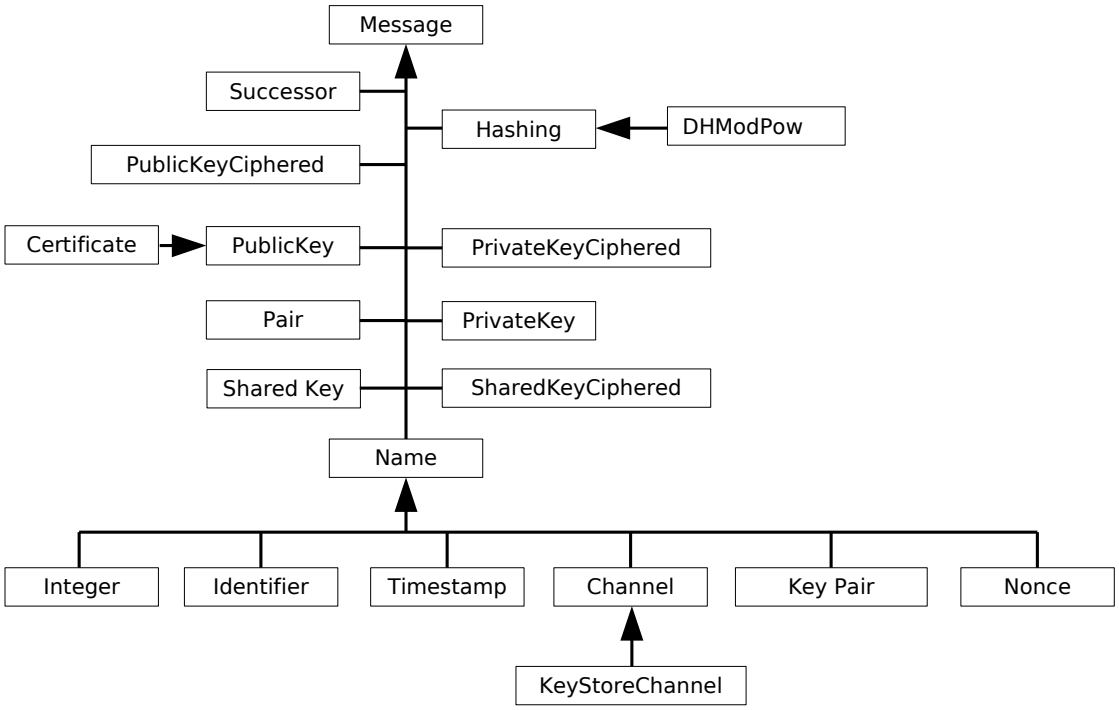

Fig. 5. SpiWrapperSim class hierarchy

```
doRun-body        ::=  statement

statement         ::=    final type id = dexpr; statement
                       | id.send(id); statement
                       | fail();
                       | event(ev-arg-list); statement
                       | if(cexpr) { statement } else { statement }
                       | { statement }
                       | return;

dexpr             ::=    new type(arg-list)
                       | method-invocation
                       | id

cexpr             ::=    method-invocation

method-invocation ::=    id.id(arg-list)
                       | id.receive(type.class)

arg-list          ::=    non-empty-arg-list
                       |

non-empty-arg-list ::=   id , non-empty-arg-list
                       | id

ev-arg-list       ::=    "name" , non-empty-arg-list
                       | "name"
```

Fig. 6. The syntax of the core language for the *doRun* method body of protocol role classes

isValid() method that returns true if the decryption has succeeded and a *getResult()* method that returns the decrypted *Message*.

The class hierarchy for data shown in Figure 5 is very simple and reflects the symbolic representation of data. Inheritance is used just for cases where a data type has several specializations. This happens, for example, for PublicKey which has Certificate as a specialization or Name, which represents an atomic or opaque data item.

The SpiWrapperSim library is extensible: for each new data model created for ProVerif, a set of corresponding classes can be created in SpiWrapperSim.

The subset of Java admitted as the core language, for the body of a doRun method in a protocol role class, is defined by means of the grammar shown in Figure 6, where nonterminals are written in italics and terminals in normal font. Nonterminals *type* and *id* are defined as valid Java identifiers, nonterminal *name* is defined as a string, while *doRun-body*, which represents the body of a doRun method, is the start symbol of this grammar.

In order to belong to the core language, the body of a doRun method must match *doRun-body*. In addition, some further static semantics restrictions apply. Nonterminal *type* must be the identifier of a SpiWrapperSim type while *id* must be the Java identifier of either a local variable, or an argument of doRun, or a method explicitly defined in one of the SpiWrapperSim classes. Further, in any method invocation taking the form *id_0.id_1(arg-list)*, id_0 must refer to a local variable or argument and id_1 must refer to one of the methods defined in the SpiWrapperSim type of id_0. Of course, all the standard Java syntax and static semantics rules must also apply, because the class as a whole must be valid Java.

As it results from this definition, the only statements admitted in the core language are:

- Local variable declarations with assignment. All variables are declared as final, i.e. they are write-once, and their value is set in their declaration. The only admitted types for variables are the SpiWrapperSim classes. The value assigned to a variable is specified by a data expression (*dexpr*), which can be either a new object of the SpiWrapperSim classes, created on the fly, or the result of a method invocation on an object belonging to one of the SpiWrapperSim classes (*method-invocation*), or a variable.
- Method invocations. Methods can be invoked only on the spiProcess object (this) or on variables, not on expressions (expressions however can be assigned to variables). Apart from method invocations that occur as data expressions inside variable declarations with assignment, it is possible to invoke the send method of a channel object, in order to send a message on that channel, and only some other methods

defined in the spiProcess class: `fail`, which has the effect of throwing an exception and terminating the current execution of the protocol session role, and `event`, which signals the occurrence of an event.
- Conditional statements with tests on the boolean values returned by methods called on the SpiWrapperSim classes (e.g. equality tests made via the *equals()* method). In order to have a direct translation into the applied π-calculus, in the core language conditional statements cannot be followed by other statements.
- Return statement, which successfully terminates the execution of the current protocol session role.

The *doRun()* method of a protocol scenario class has restrictions similar to the ones of a protocol role class. In this case, a *statement* can also take the form `start(`*non-empty-arg-list*`);` *statement*, and *type* can also be the identifier of a spiProcess class. On the other hand, calling *send*, *receive*, and *event* is not admitted. For brevity, the formal grammar is omitted for this case. The *start* method call means that, for each one of the spiProcess objects passed as arguments, an unbounded number of processes has to be started, each one executing its own *doRun()* method, with the same arguments that were passed to the spiProcess object constructor.

3.3. The Extended Specification Language

Many of the limitations of the core language are removed in the extended Java language available to the final user. A limitation that remains almost unchanged in the extended language regards the types that can be used for variables and constants (only limited use of integers and booleans is added). As already anticipated, the additional syntactic constructs introduced in the extended language can be automatically translated into functionally equivalent constructs of the core language by means of code refactoring.

This section presents the main syntax extensions of the core language and the corresponding code refactoring rules that allow their elimination. Some of them are well-known refactoring operations used in other contexts as well. As code refactoring is a well-known technique, this paper does not address the correctness of these transformations, but focuses on the soundness of the refinement process, with reference to the core language.

Here are the main extensions of the core language. The corresponding refactoring rules that map the extended constructs into the core ones are presented subsequently.

- *Integers* The extended language admits the use of integers as far as they are constants, or variables with values that can be statically determined (i.e. each use of a variable can be turned into a constant by constant propagation) or bounded counters used in loops for counting iterations.
- *List and Packet collections*: The List and Packet classes have been added to the SpiWrapperSim library in the extended language to simplify the handling of protocol messages, which have to be represented as nested pairs in the core language. A List object represents a bounded list of Message objects (all of the same type), while Packet is an abstract class that can be used as a basis to derive custom object containers, containing ordered collections of Message objects of heterogeneous types, accessible by name (this is very similar to how protocol messages are usually represented). List objects have methods that provide direct indexed access to the elements. A special message is returned when attempting to access a non-existing element. Integers can be used to specify the index of an element in the access method.
- *Bounded iterations* The applied π-calculus syntax of ProVerif does not include iteration but includes replication. However, replication generally leads to non-termination of ProVerif when the replicas have dependencies. For this reason, even though it would be possible to map Java loops (even unbounded ones) into applied π-calculus, JavaSPI supports only bounded loops in the extended language. More precisely, two different kinds of loops are admitted: counting loops (`for (int i=`*int-expr*`; i<`*int-expr*`; i++){...}`), where *int-expr* represent integer expressions whose value can be determined statically, and iterations over "List" objects (`for(`*type* `obj: `*list* `){...}`), where *type* is one of the SpiWrapperSim types and *list* is a SpiWrapperSim "List" object.
- *Separation of variable declaration from variable assignment* While the core language only admits assignment of a local variable in the same statement where the variable is declared, the extended language also admits separate declaration and assignment.
- *Non-final variables* While the core language only admits final local variables, the extended language also admits non-final variables.

- *Conditional statements without else branch* While the core language does not admit conditional statements without the else branch, the extended language does not have this limitation.
- *Code after a conditional statement* While the core language does not admit further statements after a conditional statement, the extended language does not have this limitation.
- *Custom method calls* While the core language only allows calls to methods of the SpiWrapperSim objects, the extended language also offers the user the possibility to define custom methods in order to split the code of the doRun method across various other methods. Direct or indirect recursion is not admitted, however.

Here are the refactoring operations that eliminate the extended constructs, thus mapping the extended language to the core language. They are listed in application order (refactoring rules to be applied first are listed first):

- *Elimination of custom method calls* This is a standard method inlining operation.
- *Constant propagation* Constant propagation is a standard refactoring technique. Here it is used mainly to turn integer variables and expressions into integer constants, when possible. After this step, the only integer variables that will remain in the code are the ones used in counting loops.
- *Elimination of List and Packet collections* Both List and Packet objects are transformed into sets of nested Pair objects with the same content. All the methods of these classes are turned into equivalent operations performed on the corresponding nested pair representation. As Lists may have variable length, they have a special list terminator element (a Pair composed of two special names). The presence of this special element makes it possible to test list termination using equality tests in the core language. While transforming a List object into nested pairs, loops on the List object are also unrolled (this is possible because the list length is bounded).
- *Loop unrolling* This step eliminates counting loops using a standard refactoring operation made possible by the fact that a previous constant propagation step has already turned the initial and final values of the loop into constants.
- *Turning non-final variables into final ones* A data flow analysis is performed in order to find assignments of non-final variables that occur when the assigned variable may have already been assigned. For each of these assignments, a new final variable with a new name is created and the assignment is modified so as to assign the new variable instead of the original one. Then, all subsequent uses of the old variable up to the next assignment are substituted with the new variable.
- *Elimination of code after conditional statements* The code written after a conditional statement is moved inside the conditional statement itself and, if necessary, it is duplicated in the "if" and the "else" branches. More formally, `if (cond) {` P `} else {` Q `}` $R \rightarrow$ `if (cond) {` PR `} else {` QR `}` and `if (cond) {` P `}` $R \rightarrow$ `if (cond) {` PR `} else {` R `}`.
- *Elimination of conditional statements without else branch* If there are still conditional statements without else branch, an else branch with a return statement is added to each one of them. More formally, `if (cond) {` P `}` $\rightarrow$ `if (cond) {` P `} else {` *return* `}`.
- *Joining of variable assignments with variable declarations* When this refactoring step is executed, all variables are already final (because of the application of a previous refactoring step). This guarantees that in between the declaration and the assignment of a variable there cannot be other assignments of the same variable. For this reason, it is possible to move the variable declaration into the variable assignment without altering the behavior of the program. As a variable declaration may have more than one corresponding assignment (e.g. in two different branches of a conditional statement), this refactoring step may duplicate some variable declarations.
- *Renaming of variables to make their names unique* If two local variables have the same name (in different scopes), they are renamed by appending a unique suffix to their names.

The extensions presented in this section are the ones implemented in the most recent version of JavaSPI and, we believe, the most useful ones to get adequate usability. Further extensions such as enabling the use of try-catch blocks and user-defined exceptions are possible.

```
1  public class A extends spiProcess {
2   public A(final Message...arg) {
3    super(arg);
4   }

5   @SRTypeDocument(Type=SerializationTypes.XML, ElementType="XSD_T", DescriptorPath="schema.xsd")
6   public void doRun(final Identifier plainMsg, final SharedKey sk, final Channel cAB) throws
        SpiWrapperSimException {
7    final Nonce msgNonce = new Nonce();
8    @PSecret
9    final Pair<Identifier,Nonce> msgPair = new Pair<Identifier,Nonce>(plainMsg,msgNonce);

10   @Length("16")
11   @ConcreteClass("NonceRaw")
12   @ConcretePackage("it.polito.javaSPI.custom")
13   final Nonce IV = new Nonce();

14   @Algo("AES")
15   @Iv(value="IV", type=Types.varName)
16   final SharedKeyCiphered<Pair<Identifier,Nonce>> Mk = new SharedKeyCiphered<Pair<Identifier,
         Nonce>>(msgPair,sk);

17   final Pair<SharedKeyCiphered<Pair<Identifier,Nonce>>,Nonce> p = new Pair<SharedKeyCiphered<Pair
         <Identifier,Nonce>>,Nonce>(Mk,IV);
18   cAB.send(p);


19   final Hashing x = cAB.receive(Hashing.class);

20   @Algo("SHA-256")
21   @Extends("CryptoHashing")
22   @SRType(Type=SerializationTypes.XML, ElementType="HashTypeXml")
23   final Hashing Hm = new Hashing(msgPair);
24   if (x.equals(Hm)) {
25    cAB.send(x);
26   } else fail();
27  }
28 }
```

Fig. 7. A complete JavaSPI model of a protocol actor

3.4. Annotations

Annotations are allowed in JavaSPI abstract models for two main purposes: specifying the intended security properties and providing the implementation details.

The annotations for specifying security properties must be added before invoking the Java to ProVerif generator, while the other annotations do not influence the generation of ProVerif code, hence they can be added later on, when an interoperable Java implementation must be generated. For rapid prototyping, default implementation details annotations are provided, so that concrete Java code using Java serialization can be generated directly from a pristine JavaSPI model. Annotation correctness is enforced by the JavaSPI code generators, in conjunction with the native Java support for annotations.

Figure 7 shows the annotated version of the sample protocol role specification introduced in Figure 2. Some annotations (e.g. the specification of a correspondence property) refer to the whole model and must be specified in front of the main class of the model while others (e.g. the specification of the algorithm used for encryption) refer to data objects and can be specified on a per-object basis in front of each statement that creates a data object and that defines the corresponding Java variable (i.e. `final type` *id* `= new` *type* (*arg-list*)). Annotations that refer to data objects can also be placed in front of a method or in front of a class, which affects all the objects created inside the method or class, unless the annotation is overridden by more specific annotations.

Some annotations are applicable only to some types of objects. In addition, only some values are valid. Possible violations are checked when the code is processed by the code generators.

The following subsections describe the admitted annotations. The full list of currently supported annotations is reported in Table 1. This list is extensible; for example, if new types are added to SpiWrapperSim or SpiWrapper, corresponding annotations can also be added.

3.4.1. Annotations for Specifying Security Properties

These annotations let the user specify security properties and constraints that will be given to ProVerif for verification, by including them into the automatically generated ProVerif input code. The theory developed for our framework is general enough to admit the specification and verification of any correspondence property as specified in [Bla09]. However, for simplicity, in this paper only a meaningful subset of these properties is considered. The extension to deal with all the correspondence properties supported by ProVerif is straightforward.

The `PSecret` annotation lets the user specify (weak) secrecy properties. It is applied to the data objects for which the user wants to verify secrecy. For example, the annotation at line 8 of Figure 7 specifies that `msgPair` should remain secret.

The `PVarDef` annotation is applied to data objects. It lets the user specify whether a data object has to be assumed to be public (i.e. known to the attacker) or private. In this way it is possible to define the initial attacker's knowledge.

The `PEv` and `PEvinj` annotations let the user specify non-injective and injective correspondence properties respectively. Both annotations refer to the whole model and take, as arguments, the events for which correspondence has to be verified. Each event is specified as a string, with the same syntax of event statements. Event arguments, when present, are treated as free variables. For example, the annotation `@PEvinj({"s_term","c_begin"})` means that for each occurrence of event `s_term`, a corresponding event `c_begin` occurred in the past, with the same argument values. Optionally, it is also possible to specify different ways for event argument matching, by using free variables. For example, `@PEvinj({"s_term(x,y)","c_begin(y,x)"})` means that the first argument value of `s_term` must match the second argument value of `c_begin`, and vice-versa for the other argument.

3.4.2. Annotations for Specifying Implementation Details

The annotations of this kind let the user specify parameters and features of the concrete data objects that implement the corresponding symbolic objects. All of them can be specified on a per-object basis.

For each abstract data type available in the SpiWrapperSim library, a set of implementation parameters is defined, each one with its default value. For example, for the Nonce type, the Length parameter is defined, which specifies the number of bytes of the nonce. For each of these parameters, a corresponding annotation is available, which lets the user choose the value of the parameter on a per-object basis. For example, the annotation at line 20 of Figure 7 specifies that the value to be used for the `Algo` parameter of object `Hm` is "SHA-256". This is the name of the hash algorithm to be used by the implementation to compute the cryptographic hash.

The value of a parameter specified by one of these annotations can be expressed either literally as a constant (as in line 20 of Figure 7) or indirectly as the run-time value taken by one of the protocol variables. This double possibility is enabled by using two annotation arguments to specify the parameter value: `type` and `value`. The first one can only take two possible values and specifies how the second one has to be interpreted: if `type` is set to "`value`" (which is the default), the second argument is interpreted as a literal value, while if it is set to "`varName`", the second argument is interpreted as the name of a variable whose run-time contents will be the actual parameter value. This mechanism lets one specify protocols where, for instance, the values of some parameters are negotiated at run-time by the protocol itself.

Other annotations are available to specify another feature of the concrete data objects, i.e. which concrete class has to be used to implement a symbolic data class in the implementation code.

In the JavaSPI architecture, each symbolic data class is implemented by a concrete data class available in the library. This concrete class in turn is extended by another class that adds the so-called marshaling layer, i.e. the functions used to serialize/deserialize data objects of the class, according to the protocol data encoding and encapsulation rules. The annotations `ConcretePackage` and `ConcreteClass` can be used to specify the marshaling layer classes to be used. The first annotation specifies the package that contains the classes to be used (the default value is `it.polito.spi2java.spiWrapperSR`, which is a generic

marshaling package based on Java serialization). This package must include a class for each corresponding concrete class. The second annotation, instead, specifies a single, specific class.

The `Extends` annotation is used in combination with the `ConcretePackage` annotation, in order to specify which specific class will be selected by the implementation for an object whose symbolic type has more than one corresponding concrete class in the package. This occurs, for example, in the case of Hashing, or Channel, for which different possible implementation classes exist (e.g. a Channel can be implemented by a Tcp/Ip network channel or by some other kind of communication channel).

Another possibility is to use a concrete class made of an automatically generated marshaling layer. Automatic generation is currently possible for XML and ASN.1 data types, but the generation mechanism can be extended to any other data representation system.

The `SRTypeDocument` and `SRType` annotations are used to associate objects created in the JavaSPI model with data type definitions contained in a data type description document (currently XML schema or ASN.1 description). The `SRTypeDocument` annotation specifies the type (XML or ASN.1) and location of the description document, as well as the name of the main data type to be used, and, optionally, the destination package where the marshaling class will be generated. The sample annotation shown in line 5 of Figure 7 specifies that the description document is an XML schema named "schema.xsd", while the main data type is named "XSD_T".

The `SRType` annotation can be applied to JavaSPI objects in order to specify a more specific data type to be used for those objects (which overrides the one given in the `SRTypeDocument` annotation) (see for example line 22 in Figure 7). The other information (document type and path) is inherited from the `SRTypeDocument` (line 5) annotation attached to the container object (or to the doRun method).

The marshaling layer class of an object with these annotations will be automatically generated, starting from the specified data type description document(s), along with the generation of the concrete Java code.

In Figure 7, `msgPair` has no annotation specifying its concrete class, which means it will be an instance of the default SpiWrapper class `PairSR`. That object must be kept secret to an external attacker, as specified by the `PSecret` annotation at line 8.

Lines from 10 to 12 specify that the concrete code must create a nonce (line 13), with a length of 16 bytes (line 10), as a new instance of the class `NonceRaw` (line 11), defined in the `it.polito.javaSPI.custom` package (line 12). Annotation Algo in line 14 specifies that the ciphering algorithm used for the encryption of Mk is AES, with a default key length of 128 bits. The value of the initialization vector is derived from nonce `IV`, evaluated at runtime, as specified in line 15. Therefore, the encryption operation will have all default parameters except the two (Algo and Iv) that have been overridden. In particular, the mode of encryption will be CBC, which is the default. The same procedure applies to the hashing operation in line 23, which Extends (line 21) the `CryptoHashing` class (used in cryptographic hashing operations), and uses the SHA-256 algorithm (line 20). Section 6 contains the results of Java and ProVerif generation processes applied to the example.

It is worth to remark here that, as the framework is based on a Dolev-Yao abstract model, the choices made about implementation parameters should be consistent with the usual Dolev-Yao assumptions, otherwise the security guarantees provided by the framework may not hold. In particular, any attempt to decrypt a message that is not the result of an encryption operation made with the corresponding key should fail, and the attacker should not be able to forge encrypted messages without knowing the encryption key. When using Dolev-Yao models, in order to be able to assume these two properties, it is usually assumed that non-malleable encryption schemes are used or that the message that is encrypted contains enough redundancy and the actor that performs decryption uses this redundancy to recognize the failure of the operation. In the JavaSPI framework, the implementation of decryption operations is always coupled with the subsequent unmarshaling of the result of decryption, which is necessary in order to build the Java object that contains the result of decryption. If unmarshaling fails, the decryption operation also fails and no result object is created. So, in order to be consistent with the Dolev-Yao abstraction, in JavaSPI it is enough that either a non-malleable encryption scheme is chosen or a redundant marshaling is chosen for the term that is encrypted. In our example, AES-CBC is known to be malleable, but the default marshaling layer, which is based on Java serialization, introduces redundancy, thus solving the issue.

Table 1. List of possible annotations

Annotations for specifying implementation details	
Algo, CertificateChain, ConcreteClass, ConcretePackage, ConcreteParam, DataFileName, EventsInterface, Extends, FileName, Iv, KeystoreFileName, KeystorePath, KeystoreProvider, KeystoreType, Length, Max, Min, Mode, Padding, PasswordManagerName, Provider, RemoteHost, Server, Service, Strength, Timeout	Annotations within this set specify a single value that represents a parameter used during the code generation process (see Figure 7). These annotations might contain either literal constants or variable references, and can be associated to single variable declarations or to entire methods, acting as "default" values for all the compatible objects
Annotations to automatically generate marshaling layer(s)	
SerializationLayers, SerializationLayersRoot, SerializationTypes, SRType, SRTypeDocument, UseSRLayer	Set of annotations that can be used to define and automatically generate marshaling layers (used in Figure 7).
Annotations for specifying security properties	
PAdvQuery, PConstr, PDestr, PEquation, PEv, PEvinj, PQueryList, PRule, PVarDef, PSecret	Annotations that specify security property queries and constraints used by ProVerif for verification (Section 3.4.1)

3.5. Formal Definition of JavaSPI Models

This section joins together the language elements defined so far, and provides a formalization of the whole JavaSPI modeling language, which is necessary in order to express and prove the soundness result.

A *JavaSPI abstract specification class* is a Java class that extends the *spiProcess* class, has a *doRun* method with a body written according to the core language restrictions defined in Section 3.2 for protocol roles, and may have annotations, as specified in Section 3.4.

A *JavaSPI scenario class* is a Java class that extends the *spiProcess* class, has a *doRun* method with a body written according to the core language restrictions defined in Section 3.2 for protocol scenarios, and may have annotations that specify security properties, as specified in Section 3.4.

A *JavaSPI abstract model* M_a is a pair (C, c^0), where C is a set of JavaSPI abstract specification classes (each one representing a protocol role), and $c^0 \in C$ is a JavaSPI protocol scenario class, which may have references to the other classes in C, and represents the whole model.

If a is the name of one of the JavaSPI annotations that can be applied to single objects (e.g. *Algo*, or *PSecret*), and x is the name of a variable occurring in the *doRun* method of a class $c \in C$, the function $a(x)$ represents the value of annotation a for variable x. If annotation a is not present for x, $a(x)$ is the default value. For annotations that have no value (e.g. *PSecret*), $a(x)$ conventionally denotes the boolean value true if the annotation is present and the value false if it is not (or, in other words, false is the implied default value). When the class c to which $a(x)$ refers is not obvious, the class is specified as a subscript of a.

A *JavaSPI concrete implementation class* is a Java class automatically generated by the JavaSPI framework from a JavaSPI abstract specification class. Formally, the deterministic behavior of this automatic generation process is represented by a function $J()$ that maps a JavaSPI abstract specification class c into its corresponding concrete implementation class $J(c)$. This function will be defined formally in Section 6.

A *JavaSPI refined model* M_r is similar to an abstract model, but some of the elements of C are JavaSPI concrete implementation classes rather than abstract specification classes.

A JavaSPI abstract model $M_a = (C_a, c^0)$ can be refined into a JavaSPI refined model $M_r = (C_r, c^0)$ by turning one or more of the classes in C_a into their corresponding JavaSPI concrete implementation classes.

A *JavaSPI refinement* from M_a to M_r is formally represented by a function $r : C_a \to C_r$ such that, for each $c \in C_a$, either $r(c) = c$ or $r(c) = J(c)$. The refinement function r is then lifted to abstract models in the obvious way: $r(C_a, c^0) = (\{r(c) | c \in C_a\}, c^0)$.

Note that only single roles are refined, while c^0, which represents the structure of the whole model, is never refined.

Table 2. ProVerif syntax used to generate model algorithms

0	Process termination
$!P$	Process replication
$P\|Q$	Parallel composition
$new\ a; P$	Restriction (a is a fresh name in P)
$out(N, M); P$	Output of message M on channel N
$in(N, x); P$	Input of a message from channel N, stored in x
$let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$	Destructor application: if g succeeds continues as P else as Q
$let\ x = M\ inP$	Assignment
$if\ M = N\ then\ P\ else\ Q$	Equality test
event $a(M); P$	Event with name a and data M
$let\ x$ suchthat member: $x, S\ in\ P$	Non-deterministic assignment of x with term belonging to S

4. Formally Proved Security With JavaSPI

In the next sections, the JavaSPI modeling language is given formal semantics by defining a function that translates JavaSPI models into the applied π-calculus accepted by ProVerif, which already has formal semantics. That is, the formal semantics of the Java modeling language constructs is determined by the applied π-calculus constructs they are mapped to. In this way, formal verification of security properties under the Dolev-Yao assumptions becomes possible by formally verifying the applied π-calculus resulting from the translation. Of course, the implementation of the SpiWrapperSim classes has been built in such a way that the simulated behavior reproduces the applied π-calculus semantics. Proving the correctness of this simulation in the SpiWrapperSim classes is not so important as it does not have impact on the soundness of the implementation code generation, as explained later on. For this reason it is left out.

In order to be able to formally prove the soundness of the refinement procedure that leads from JavaSPI abstract models to their refinements, a formal semantics has to be given to the obtained concrete Java implementations too. This will be done in the next sections, coherently with one of the already existing formalizations of the Java language semantics. Then, having formal semantics for both the abstract models and the derived Java implementations, it becomes possible to formally relate them. Our final aim is to formally prove that, at least for an important class of security properties, the JavaSPI refinement process is sound. The soundness concept we refer to can be informally expressed in this way: if a JavaSPI model satisfies a security property, then a refinement of that model, obtained by substituting one or more protocol roles with their corresponding Java implementations according to the JavaSPI refinement process, satisfies the same security property. More formally, let M_a be a JavaSPI abstract model, let ϕ_a be a security property referred to M_a, and let r be a JavaSPI refinement. The soundness theorem can then be formulated as $(M_a \models \phi_a) \Rightarrow (r(M_a) \models \phi_a)$ where the notation $M \models \phi$ means as usual that model M satisfies property ϕ.

The following sections provide the formal definitions that are necessary in order to formally state and prove the soundness theorem.

5. The Java-ProVerif Translation

As JavaSPI and the applied π-calculus accepted by ProVerif share the same abstraction level and the same modeling approach, the translation from a JavaSPI model to a model accepted by ProVerif is straightforward: it basically consists in a syntactical transformation. This transformation is formally specified by a translation function $PV()$, which is initially defined on identifiers and then progressively lifted to statements of the JavaSPI core language and finally to JavaSPI abstract models.

The portion of the applied π-calculus generated by $PV()$ has a syntax including terms and processes. The syntax for terms is specified by the production $M ::= x \mid a \mid f(M_1, \ldots, M_n)$, where M ranges over terms, x ranges over variables, a ranges over names, and f ranges over constructor names. The syntax for processes is shown in Table 3 where P and Q range over processes, M and N range over terms, g ranges over destructor names, and S ranges over sets of terms.

An applied π-calculus model accepted by ProVerif can be represented by a pair (P, W) where P is a process, and W is a set of equations that describe the behavior of the destructors occurring in P.

For a Java identifier id, $PV(id)$ is the ProVerif name, variable, or function (constructor or destructor) that corresponds to id. If applied to *arg-list* or *non-empty-arg-list*, the function returns the result of sequentially applying $PV()$ to each element of the list, keeping comma separators as in the original list. Similarly, for a SpiWrapperSim type *type*, $PV(type)$ is the ProVerif identifier for the corresponding constructor. For a JavaSPI statement of the core language, and for all the other syntactic constructs defined in Figure 6, such as *dexpr* and *method-invocation*, the formal definition of PV is given by the rules specified in Table 3. As usual, the first rule that matches is applied. In Table 3, $L()$ combines its arguments (possibly empty comma separated lists of strings) into a single list of comma-separated strings, while $PFun(id_0, id_1)$ represents the name of the ProVerif function (constructor or destructor) that corresponds to method id_1 of the JavaSPI object id_0. For example, if id_0 is a SharedKeyCiphered object, then $PFun(id_0, \text{decrypt})$ is `SymDecrypt`.

In order to achieve the soundness result we aim at, it is necessary to consider that some operations (e.g. the execution of a destructor such as a decryption operation) may fail in the concrete Java implementation for reasons that are not represented in the abstract model. For example, a decryption operation may fail because the algorithm being used for decryption is not the same that was used for encryption. As the information about the decryption algorithm is added in the refinement process, in the form of annotation, it is not part of the abstract model, and in the abstract model a decryption always succeeds if the right key is used for decryption, independently of the algorithm being used in the implementation. This difference of behavior between the abstract model and the implementation may lead to a disruption of soundness. For example, let us consider a protocol that executes a particular sequence of actions on failure of a decryption operation. If the decryption operation may fail in the implementation but not in the abstract model, the execution of that sequence of actions may occur in the implementation but not in the abstract model. If that sequence of actions leads to the violation of a security property which is otherwise always satisfied, such violation could occur in the implementation but not in the abstract model, where that sequence of actions can never be executed.

In order to avoid this issue, we modify the usual ProVerif model of the operations (such as decryption) that may fail because of implementation details, by introducing a nondeterministic choice: the operation may nondeterministically fail in the abstract model, thus taking into account the possibility of failure that exists in the implementation.

This can be seen in Table 3: the translation of a statement that takes the form "final *type* $id_2 = id_0.id_1$(*arg-list*);*statement*" is
let x suchthat member:x,true_false_set in
let $PV(id_2) = PFun(id_0, id_1)(L(PV(id_0), PV(\textit{arg-list}), x))$ in $PV(\textit{statement})$

This means that x is set nondeterministically to true or false (true_false_set is a set that includes only true and false). The value of x is then passed as an additional argument to the ProVerif function that represents method id_1 of object id_0. This function fails (i.e. stops the process) if the passed x is true or behaves normally if x is false. For example, the ProVerif destructor that models symmetric key decryption is defined by the rule

```
reduc SymDecrypt(SymEncrypt(M,Key), Key, false) = M.
```

The case when the function fails represents the case of a failure of the corresponding operation in the implementation because of wrong parameters.

Note that this issue does not regard send and receive operations because their failure in the Java implementation code always leads to throwing a SpiWrapper exception which immediately aborts the process. So, for send and receive, the modeling of arbitrary failures is not strictly necessary in order to achieve soundness.

The *doRun()* method of a protocol scenario class is translated using the same rules used for protocol role classes, with the following additional rule that specifies how the *start()* method calls are translated. The additional rule is:

$$PV(\text{start}(id_1, \ldots, id_n);) \rightarrow (\ (\ !\ PV(DoRun(id_1))\ )\ |\ \ldots\ |(\ !\ PV(DoRun(id_n))\ )\ )$$

where $id_1, \ldots, id_n$ are variables holding *spiProcess* objects and the $DoRun(id)$ function returns the Java code of the $doRun()$ method body of the class of id, with its formal arguments substituted by the arguments that were passed to the constructor of the id object. As each id object is created by a new statement that calls the corresponding constructor with the actual arguments, $DoRun(id)$ can be computed statically.

In practice, for each argument of the *start* call, an applied π-calculus process is created with replication,

Table 3. Definition of the $PV()$ translation function for the core language defined in Figure 6.

PV(final *type* *id* = new *type*(*arg-list*); *statement*)	→	new $PV(id)$; $PV(statement)$	if *type* is a *Name* and PVarDef(*id*)=PRIVATE
PV(final *type* *id* = new *type*(*arg-list*); *statement*)	→	let $PV(id) = PV(type)(PV(\text{arg-list}))$ in $PV(statement)$	
PV(final *type* id_1 = id_0.receive(*type*.class); *statement*)	→	let x suchthat member:x,true_false_set in if (x=false) then in$(PV(id_0), \overline{PV(id_1)})$; $PV(statement)$	
PV(final *type* id_2 = $id_0.id_1$(*arg-list*);*statement*)	→	let x suchthat member:x,true_false_set in let $PV(id_2) = \mathcal{F}(L(PV(id_0), \overline{PV(\text{arg-list})}, x))$ in $PV(statement)$	where $\mathcal{F}$=$PFun(id_0, id_1)$
PV(final *type* id_1 = id_0;*statement*)	→	let $PV(id_1) = PV(id_0)$ in $PV(statement)$	
$PV(id_0.\text{send}(id_1);statement)$	→	out$(PV(id_0), PV(id_1))$; $PV(statement)$	
$PV(fail())$	→	0	
PV(event("*name*", *arg-list*)); *statement*)	→	event *name*(PV(*arg-list*)); $PV(statement)$	
PV(if(*cexpr*){*statement*$_1$} else{*statement*$_2$})	→	if $PV(cexpr)$ then ($PV(statement_1)$) else ($PV(statement_2)$)	
PV(return;)	→	0	
$PV(id_0.id_1\ (arg\text{-}list))$	→	$\mathcal{F}(L(PV(id_0), PV(arg\text{-}list))) = \text{true}$	where $\mathcal{F}$=$PFun(id_0, id_1)$

which means that an unbounded number of instances of that process may be created, and all the instances are combined in parallel.

The JavaSPI-ProVerif translation of an abstract model $M_a = (C_a, c^0)$ is the applied π-calculus model $PV((C_a, c^0)) = (PV(c^0), W)$, where W is a fixed set of equations defining the behavior of all the destructors available in the JavaSPI library and $PV(c^0) = PV(DoRun(c^0))$, i.e. the translation of the body of the *DoRun* method of c^0.

In Table 3, the translation of Java annotations that specify security properties has been omitted for simplicity. These annotations are translated into corresponding ProVerif queries. More precisely, if the Java variable identified by x is annotated with the PSecret annotation, the ProVerif query `query attacker:`x is automatically generated and included into the file to be given to ProVerif for verification. Similarly, the annotations taking the form $PEv(\text{“}event1(arg\text{-}list_1)\text{”}, \text{“}event2(arg\text{-}list_2)\text{”})$ are translated into the queries `query ev:evt_`$event1$`(`$arg\text{-}list_1$`) ==> ev:evt_`$event2$`(`$arg\text{-}list_2$`)`.

6. The JavaSPI Refinement Process

The Java implementation code generated from an abstract Java model is based on the SpiWrapper library, which provides the concrete data classes that replace the abstract ones and that include implementations of the cryptographic and communication operations in the Spi2Java framework. This choice simplifies the translation of the abstract model into a concrete implementation. The result is that even the syntax used in the two codes is very similar, as shown in the example in Figure 8: the main differences are that the two codes use different classes for representing data, and the concrete code includes implementation details derived from the annotations of the abstract model (previously presented in Figure 7). Note that SpiWrapper and SpiWrapperSim share the same class names. For example, SharedKey in Figure 8 is the concrete SpiWrapper class corresponding to the abstract SpiWrapperSim class.

The automatic translation from an abstract specification class to a concrete implementation class is formally specified by a function $J()$. First, $J()$ is defined on the domain of statements of the core language in Table 4, where the definition of $J()$ is given also for the other nonterminals of the core language syntax, such as *dexpr* and *method-invocation*. In Table 4, as usual, the first rule that matches, from top to bottom, is applied.

The annotation value ConcreteClass(*id*), which specifies the type to be used for variable *id* in the concrete implementation, defaults to the class based on Java serialization that is provided by the framework for

Java concrete implementation

```
public class A_Protocol {
 public void doRun(Identifier plainMsg, SharedKey sk, Channel cAB) throws SpiWrapperException {
  try {
   final Nonce msgNonce = new NonceSR("8");
   final Pair msgPair = new PairSR(plainMsg, msgNonce);
   final Nonce IV = new NonceRaw("16");
   final SharedKeyCiphered Mk = new SharedKeyCipheredSR(msgPair, sk, "AES", IV.toString(), "CBC",
        "PKCS5Padding", "SunJCE");
   final Pair p = new PairSR(Mk, IV);
   cAB.send(p);
   final Hashing x = (Hashing)cAB.receive(new CryptoHashingSR());
   final Hashing Hm = new CryptoHashingSR(msgPair, "SHA-256", "SUN");
   if (x.equals(Hm)){
    cAB.send(x);
   } else {
    throw new SpiWrapperException("check failed");
   }
   return;
  } catch (Throwable t) {
   /* cleanup, if necessary */
   throw t;
  } finally {
   /* cleanup, if necessary */
  }
 }
}
```

ProVerif model

```
data cAB_1/0. (* communication channel *)
...
query attacker:(plainMsg_1,msgNonce). (* secrecy query *)
...
process
let truefalseset00 = consset (true, consset (false, emptyset)) in
new plainMsg_1;
new n;
let sk_2 = SharedKey(n) in (* the shared key term *)
(
 (!
  new msgNonce;
  let msgPair = (plainMsg_1,msgNonce) in (* the secret pair message *)
  new IV;
  let Mk = SymEncrypt(msgPair,sk_2) in
  let p = (Mk,IV) in
  out(cAB_1,p);
  in(cAB_1,x);
  let Hm = H(msgPair) in
  if x = Hm then (
   out(cAB_1,x);
   0
  )
 ) |... (* other actors of the model *)
```

Fig. 8. An excerpt of how the abstract model in Figure 7 is converted into the corresponding concrete Java implementation and ProVerif specification (comments have been added manually)

Table 4. Definition of the $J()$ translation function

J(final *type* id = new *type*(*arg-list*); *statement*)	$\rightarrow$	final *typeC* id = new *typeC*(ConcrArg(*arg-list*, id)); J(*statement*)	where *typeC*=ConcreteClass(id)
J(final *type* id_1 = id_0.receive(*type*.class); *statement*)	$\rightarrow$	final *typeC* $id_1 = id_0$.receive(*typeC*.class); J(*statement*)	where *typeC* = ConcreteClass(id_1)
J(final *type* id = *dexpr*; *statement*)	$\rightarrow$	final *typeC* $id = J$(*dexpr*); J(*statement*)	where *typeC*=ConcreteClass(id)
$J(id_0$.send(id_1); *statement*)	$\rightarrow$	id_0.send(id_1); J(*statement*)	
$J(id_0.id_1$(*arg-list*))	$\rightarrow$	$id_0.id_1$(ConcrArg(*arg-list*, id_0, id_1))	
$J(id)$	$\rightarrow$	id	
J(fail();)	$\rightarrow$	throw new SpiWrapperException();	
J(event(*ev-arg-list*)); *statement*)	$\rightarrow$	event(*ev-arg-list*); J(*statement*)	
J(if(*cexpr*){*statement*$_1$} else{*statement*$_2$})	$\rightarrow$	if(J(*cexpr*)){ J(*statement*$_1$) } else { J(*statement*$_2$) }	
J(return;)	$\rightarrow$	return;	

each data type or, if ConcretePackage(id) is defined, it defaults to the class available in that package. The function ConcrArg(*arg-list*, id) computes the list of arguments to be passed to the concrete constructor called for setting variable id, from the argument list *arg-list* that was used for the abstract constructor. ConcrArg(*arg-list*, id) is the concatenation of *arg-list* with the (possibly empty) list of extra arguments coming from the annotations of id. For example, let us consider variable IV defined in Figure 7. For that variable we have ConcreteClass(IV)="NonceRaw" and ConcrArg("",IV)="16", because the Nonce abstract constructor has an empty list of arguments and the concrete constructor expects one extra argument whose value is given by the annotation Length(IV). The resulting call of the concrete constructor is shown in Figure 8.

Similarly, ConcrArg(*arg-list*,id_0,id_1) computes the list of arguments to be passed when calling the concrete method id_1 on the object variable id_0, where *arg-list* is the list of arguments occurring in the call of the corresponding abstract method.

Having defined how $J()$ transforms the statements of the core language, J is lifted to be applied to *doRun* methods as follows. Let *ArgList*(m) and *body*(m) be respectively the argument list and the body of a *doRun* method m. Then, $J(m)$ is defined as

```
1 public void doRun(ArgList(m)) throws SpiWrapperException {
2   try {
3       J(body(m))
4     } catch (Throwable t) {
5    /* cleanup, if necessary */
6    throw t;
7   } finally {
8    /* cleanup, if necessary */
9   }
```

Finally, the lifting of J to an abstract specification class c^a is defined as a function $J(c^a)$ that replaces the *doRun* method of c^a, $DoRun(c^a)$, with its transformation $J(DoRun(c^a))$, eliminates the "extends SpiProcess" clause in the class header and adds an abstract *event(Message...)* method to the class. This last method has to be implemented in order to manage the outputs of a protocol session (outputs are actually events generated during protocol execution).

7. Notation and Formal Semantics

In this section we give the formal semantics of the different representations presented in the previous sections, but first we introduce some formal notation and concepts (Labelled Transition Systems and weak simulation) that will be used for the formulation of the formal semantics and for the soundness proof.

Table 5. Definition of the reduction relation for the applied π-calculus

$(\mathcal{E}, \mathcal{P} \cup \{0\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P})$
$(\mathcal{E}, \mathcal{P} \cup \{!P\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{P, !P\}$
$(\mathcal{E}, \mathcal{P} \cup \{P \vert Q\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{P, Q\})$
$(\mathcal{E}, \mathcal{P} \cup \{new\ a; P\})$	$\rightarrow$	$(\mathcal{E} \cup \{a'\}, \mathcal{P} \cup \{P[a'/a]\})$ where $a' \notin \mathcal{E}$
$(\mathcal{E}, \mathcal{P} \cup \{out(N, M); Q,\ in(N, x); P\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{Q, P[M/x]\})$
$(\mathcal{E}, \mathcal{P} \cup \{let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{P[M'/x]\})$ if $g(M_1, \ldots, M_n) \rightarrow M'$
$(\mathcal{E}, \mathcal{P} \cup \{let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{Q\})$ if $\nexists M'.\ g(M_1, \ldots, M_n) \rightarrow M'$
$(\mathcal{E}, \mathcal{P} \cup \{let\ x = M\ in\ P\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{P[M/x]\})$
$(\mathcal{E}, \mathcal{P} \cup \{if\ M = N\ then\ P\ else\ Q\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{P\})$ if $M = N$
$(\mathcal{E}, \mathcal{P} \cup \{if\ M = N\ then\ P\ else\ Q\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{Q\})$ if $M \neq N$
$(\mathcal{E}, \mathcal{P} \cup \{\text{event } a(M); P\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{P\})$
$(\mathcal{E}, \mathcal{P} \cup \{let\ x \text{ suchthat member: } x, S\ in\ P\})$	$\rightarrow$	$(\mathcal{E}, \mathcal{P} \cup \{P[M/x]\})$ if $M \in S$

7.1. Notation

Definition 1 (Labelled Transition System (LTS)). A Labelled Transition System (LTS) is a quadruple $(\Sigma, s_0, \Lambda, \rho)$ where Σ is a set of states, $s_0 \in \Sigma$ is the initial state, Λ is a set of labels representing events associated with state transitions and $\rho \subseteq \Sigma \times \Lambda \times \Sigma$ is the transition relation. A special label $\tau \in \Lambda$ is used to denote unobservable events.

In the sequel, s ranges over Σ and λ ranges over Λ. As usual, $s \xrightarrow{\lambda} s'$ is an alternative notation for $\rho(s, \lambda, s')$, which means that a transition from s to s' labeled by λ is possible, while $s \xrightarrow{\lambda}$ means that $s \xrightarrow{\lambda} s'$ for some s', and $s \not\xrightarrow{\lambda}$ is its negation. Moreover, $\xrightarrow{\lambda_1,\cdots,\lambda_n} = \xrightarrow{\lambda 1} \cdots \xrightarrow{\lambda_n}$ denotes the composition of relations $\xrightarrow{\lambda 1},..,\xrightarrow{\lambda_n}$, while $\stackrel{\tau}{\Rightarrow} = \xrightarrow{\tau}{}^*$ is the transitive closure of $\xrightarrow{\tau}$ and, for $\lambda \neq \tau$, $\stackrel{\lambda}{\Rightarrow} = \stackrel{\tau}{\Rightarrow}\xrightarrow{\lambda}\stackrel{\tau}{\Rightarrow}$. Finally, $\stackrel{\lambda_1,\cdots,\lambda_n}{\Rightarrow} = \stackrel{\lambda 1}{\Rightarrow} \cdots \stackrel{\lambda_n}{\Rightarrow}$.

Definition 2 (Traces). The traces of an LTS $\mathcal{L} = (\Sigma, s_0, \Lambda, \rho)$ are defined as

$$traces(\mathcal{L}) = \bigcup_{n \in \mathbb{N} - \{0\}} n\text{-}traces(\mathcal{L}) \quad \text{where} \quad n\text{-}traces((\Sigma, s_0, \Lambda, \rho)) = \{\lambda_1 \cdots \lambda_n | s_0 \xrightarrow{\lambda_1,\cdots,\lambda_n}\}.$$

Definition 3 (Observable traces). The notation $otraces(\mathcal{L})$ is used to denote the observable traces of $\mathcal{L}$, i.e. the projections of the traces of $\mathcal{L}$ onto observable events only. Formally, $otraces(\mathcal{L}) = \{t \backslash \{\tau\} | t \in traces(\mathcal{L})\}$ where $t \backslash L$ is the trace obtained by stripping out of t the events in L, i.e. $(\lambda t) \backslash L = \lambda(t \backslash L)$ if $\lambda \notin L$ and $(\lambda t) \backslash L = t \backslash L$ if $\lambda \in L$.

Definition 4 (Weak simulation). Given two LTSs $\mathcal{L}_1 = (\Sigma_1, s_{01}, \Lambda, \rho_1)$ and $\mathcal{L}_2 = (\Sigma_2, s_{02}, \Lambda, \rho_2)$, a relation $sim \subseteq \Sigma_1 \times \Sigma_2$ is a weak simulation if and only if $\forall (s_1, s_2) \in sim. \forall \lambda. \forall s_1'$

$$s_1 \xrightarrow{\lambda} s_1' \implies \exists s_2'. (s_2 \stackrel{\lambda}{\Rightarrow} s_2' \ \wedge \ (s_1', s_2') \in sim). \tag{1}$$

Definition 5 (Weak simulation of LTSs). Given two LTSs $\mathcal{L}_1 = (\Sigma_1, s_{01}, \Lambda, \rho_1)$ and $\mathcal{L}_2 = (\Sigma_2, s_{02}, \Lambda, \rho_2)$, $\mathcal{L}_1$ *weakly simulates* $\mathcal{L}_2$ iff there exists a weak simulation relation sim such that $(s_{01}, s_{02}) \in sim$.

7.2. Operational semantics of the applied π-calculus

The operational semantics of the applied π-calculus accepted by ProVerif has been formally defined in [Bla09] by means of a reduction relation $\rightarrow \subseteq S^2$, where S is the set of states. The state of a closed set of concurrent processes is represented as a configuration $(\mathcal{E}, \mathcal{P})$, where $\mathcal{P}$ is a multiset including the concurrent processes and $\mathcal{E}$ is a set of names that are free in the processes that belong to $\mathcal{P}$. Table 5 recalls the definition of this reduction relation.

7.3. Operational semantics of JavaSPI processes

For the Java language, a formal semantics covering a significant subset of Java, known as "Middleweight Java" (MJ), has been defined in [BPP03]. In MJ, the snapshot of a Java program execution in a precise instant is represented by a configuration, which essentially is composed of the sequence of code statements that still have to be executed, and the representation of the state of the memory the program has allocated so far (heap objects and contents of variables that are in the current scope). A reduction relation specifies how these configurations can evolve.

Both the core language used to specify JavaSPI abstract models and the concrete Java implementations generated by the JavaSPI framework are almost completely covered by the MJ Java subset. Only the management of exceptions is not fully covered by MJ, because MJ does not include throw and try-catch statements, and its semantics just considers two classes of exceptions, i.e. NullPointerException and ClassCastException.

As JavaSPI implementations make a very limited use of exceptions, a very slight extension to MJ would be enough to represent their semantics. However, as our aim is to relate the execution of a Java program with the execution of an applied π-calculus process, a representation of the Java implementation semantics less detailed than the MJ one is enough. More precisely, we do not need the exact representation of the memory state of the Java program, but we just need to keep track of how the memory state of the Java program is related to the symbolic terms of the applied π-calculus model. Also, for what concerns the execution of cryptographic and communication operations in the Java implementation, we assume they are implemented correctly and, hence, we are not interested in analyzing the execution of their implementation code. Instead, we consider the execution of each one of these operations as a single execution step in the evolution of the Java program configurations. For this purpose, in the next subsection we define a semantics for JavaSPI implementations, which is strictly related to the MJ semantics, but is less detailed, and incorporates a number of assumptions that we make about cryptographic and communication operations. More precisely, our semantics represents the evolution of the Java implementation under the assumption that cryptography is perfect, as in the Dolev-Yao model.

A configuration of a JavaSPI process representing a JavaSPI role is a pair (S, σ), where S is the core language statement ready to be executed (here we consider that a statement also includes all the next statements, in analogy with what has been done for the syntax of the core language), and σ is a partial function that maps each Java variable that has been initialized so far onto the data term of the applied π-calculus that symbolically represents the value assigned to the variable. For example, $\sigma(x) = SymEncrypt(M, key)$ means that variable x has been assigned the result of encrypting the message symbolically represented by M with the key symbolically represented by key. Function σ is lifted to lists of variables as follows:

$$\sigma(id, arg\text{-}list) = \begin{cases} id, \sigma(arg\text{-}list) & \text{if } id \notin dom(\sigma) \\ \sigma(id)\sigma(arg\text{-}list) & \text{otherwise} \end{cases}$$

The generated Java code shares with the core language the use of write-once variables and the absence of custom method calls. Moreover, both in the core language and in the generated Java code, even if multiple scopes can be defined by means of if/else instructions, variables are required to have unique names within the whole model (it is possible to use the same variable names in different scopes only in the extended language). For this reason, in each role implementation program each variable can be uniquely identified by its name, and the tracking of scopes is not necessary.

Note that, for simplicity, even if the code of the method that executes a protocol role in a concrete implementation class includes an outer try-catch block used to intercept all exceptions, we represent in S only the contents of the try branch of this block, while the existence of the enclosing try-catch block is assumed implicitly when defining the evolution of the configurations. This is possible because the try-catch block is fixed.

After having defined configurations, we have to specify how they evolve during the execution of a JavaSPI program. Table 6 presents the evolution rules, defined coherently with the MJ semantics, where α is the function that maps each concrete class $typeC$ onto its corresponding abstract class $type$, belonging to the SpiWrapperSim library, $AbsArg(arg\text{-}list_c, id)$ is the function that maps a concrete argument list $arg\text{-}list_c$ for the constructor of the concrete class of variable id onto the corresponding abstract one, by stripping the extra parameters, and $AbsArg(arg\text{-}list_c, id_0, id_1)$ is the similar function for the arguments of method id_1 of the class of variable id_0. gnd is the function that maps a set of symbolic terms onto the set of their ground

Table 6. Evolution rules for the JavaSPI process semantics

(final $typeC$ id = new $typeC(arg\text{-}list_c)$; $stat, \sigma$)	$\rightarrow$	$(stat, \sigma \cup \{(id, type(\sigma(arg\text{-}list_a)))\})$	where $type = \alpha(typeC)$ and $type$ is not a $Name$ and $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id)$
(final $typeC$ id = new $typeC(arg\text{-}list_c)$; $stat, \sigma$)	$\rightarrow$	$(stat, \sigma \cup \{(id, M)\})$	where $M \notin gnd(Im(\sigma))$ and $\alpha(typeC)$ is a $Name$
(final $typeC$ $id_1 = id_0$.receive($typeC$.class); $stat, \sigma$)	$\rightarrow$	$(stat, \sigma \cup \{(id_1, type(X_1, \ldots, X_n))\})$	where $type = \alpha(typeC)$ and $n = arity(type)$ and $X_i \notin gnd(Im(\sigma))$
(final $typeC$ $id_1 = id_0.id_2(arg\text{-}list_c); stat, \sigma$)	$\rightarrow$	$(stat, \sigma \cup \{(id_1, M)\})$	if PFun$(id_0, id_2)(\sigma(L(id_0, arg\text{-}list_a)))$ $\rightarrow M$ and $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id_0, id_2)$
(final $typeC$ $id_1 = id_0; stat, \sigma$)	$\rightarrow$	$(stat, \sigma \cup \{(id_1, \sigma(id_0))\})$	
$(id_0$.send$(id_1); stat, \sigma)$	$\rightarrow$	$(stat, \sigma)$	
(event$(ev\text{-}arg\text{-}list_c); stat, \sigma)$	$\rightarrow$	$(stat, \sigma)$	
(if$(id_0.id_1(arg\text{-}list_c))\{stat_1\}$ else$\{stat_2\}, \sigma)$	$\rightarrow$	$(stat_1, \sigma)$	if PFun$(id_0, id_1)(\sigma(L(id_0, arg\text{-}list_a)))$ $\rightarrow true$ and $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id_0, id_1)$
(if$(id_0.id_1(arg\text{-}list_c))\{stat_1\}$ else$\{stat_2\}, \sigma)$	$\rightarrow$	$(stat_2, \sigma)$	if PFun$(id_0, id_1)(\sigma(L(arg\text{-}list_a, id_0)))$ $\rightarrow false$ and $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id_0, id_1)$
$(stat, \sigma)$	$\rightarrow$	(THR, σ)	

terms (i.e. terms not composed of other terms), and L is the function that concatenates two possibly empty argument lists into a single one.

In the generated Java code, two classes of exceptions are possible: those belonging to the SpiWrapperException class (e.g. when one of the cryptographic or networking operations fails) and runtime exceptions. In particular, a runtime exception may be thrown when any instruction in the generated Java code is executed (e.g. due to the exhaustion of available memory). The possible occurrence of one of these exceptions (i.e. subclasses of Java Throwable class) is represented, in analogy to what MJ does for NullPointerException and ClassCastException, with evolution rules that lead to the special configurations (THR, σ), which have no further possible evolutions. These rules correctly represent the behavior of the Java implementation code when an exception is thrown, because each configuration includes the implicit try-catch statement that encloses S, and stops the execution of the protocol session in the catch branch. Hence the exceptions generated in a protocol role implementation process always stop the execution of that process, as represented in the semantics. The correctness of the semantics presented in Table 6 with respect to MJ or other Java semantics is not proved in this paper, but it is assumed.

Having formal semantics for applied π-calculus and for the generated Java code, it is now possible to give formal semantics to refined models too. This is done by defining a labeled transition system (LTS) that represents the evolution of a refined model, where labels are introduced in order to be able to distinguish observable events from unobservable ones, as explained in the next section.

In order to define the semantics of refined models, the applied π-calculus syntax is first slightly extended. A new process, denoted by $j(S, \sigma)$, is added to the set of applied π-calculus processes. $j(S, \sigma)$ represents the Java process corresponding to the configuration (S, σ). The free names of $j(S, \sigma)$ are defined as $fn(j(S, \sigma)) = gnd(Im(\sigma))$.

In the sequel, R denotes an extended applied π-calculus process, while $\mathcal{R}$ denotes a set of such processes.

The following labeled transition relations are first defined on pairs of extended applied π-calculus processes. $R \stackrel{c!M}{\rightarrow} R'$ means that R can output the symbolic message M on the symbolic channel c and evolve into R'. Similarly, $R \stackrel{c?M}{\rightarrow} R'$ means that R can input the symbolic message M on the symbolic channel c and evolve into R', $R \stackrel{\nu M}{\rightarrow} R'$ means that R can create a fresh name M and evolve into R', and $R \stackrel{event(n,M)}{\rightarrow} R'$ means that R can generate event $n(M)$, where n is the event name and M is a symbolic term, and evolve into R'. Finally, $R \stackrel{\tau}{\rightarrow} R'$ means that R can evolve into R' by performing an internal step which is neither the creation of a fresh name nor the generation of an event.

Table 7. Definition of labelled reduction for processes that do not take the form $j(S,\sigma)$

$$
\begin{aligned}
R \xrightarrow{c!M} R' &\iff R = out(c,M);R' \\
R \xrightarrow{c?M} R' &\iff R = in(c,x);R'' \text{ with } R' = R''[M/x] \\
R \xrightarrow{\nu M} R' &\iff R = new\ x;R'' \text{ with } R' = R''[M/x] \\
R \xrightarrow{event(n,M)} R' &\iff R = event\ n(M);R' \\
R \xrightarrow{\tau} R' &\iff \exists \mathcal{E}.((\mathcal{E},\{R\}) \rightarrow (\mathcal{E},\{R'\})) \text{ and } \nexists M,n.(R \xrightarrow{\nu M} R' \vee R \xrightarrow{event(n,M)} R')
\end{aligned}
$$

Table 8. Definition of labelled reduction for processes that take the form $j(S,\sigma)$

$$
\begin{aligned}
j(S,\sigma) \xrightarrow{c!M} j(S',\sigma') &\iff S = id_0.send(id_1);S' \text{ for some } id_0 \text{ and } id_1, \text{ with } \sigma(id_0) = c, \text{ and } \sigma(id_1) = M, \text{ and } \sigma' = \sigma \\
j(S,\sigma) \xrightarrow{c?M} j(S',\sigma') &\iff S = final\ typeC\ id_1 = id_0.receive(typeC.class);S', \text{ for some } id_0,\ id_1 \text{ and } typeC, \text{ with } \sigma(id_0) = c, \text{ and } \sigma' = \sigma \cup \{(id_1,M)\} \\
j(S,\sigma) \xrightarrow{\nu M} j(S',\sigma') &\iff S = \text{final } type\ id = \text{new } type(arg\text{-}list);S', \text{ for some } type,\ id, \text{ and } arg\text{-}list \text{ such that } type \text{ is a } Name \text{ and PVarDef}(id)\text{=PRIVATE, and } \sigma' = \sigma \cup \{(id,M)\} \\
j(S,\sigma) \xrightarrow{event(n,M)} j(S',\sigma') &\iff S = event(ev\text{-}arg\text{-}list);S', \text{ for some } ev\text{-}arg\text{-}list \text{ such that } \sigma(ev\text{-}arg\text{-}list) = L(n,M), \text{ and } \sigma' = \sigma \\
j(S,\sigma) \xrightarrow{\tau} j(S',\sigma') &\iff (S,\sigma) \rightarrow (S',\sigma') \text{ according to the JavaSPI semantics and } \forall \lambda \neq \tau.\ j(S,\sigma) \not\xrightarrow{\lambda} j(S',\sigma')
\end{aligned}
$$

If R does not take the form $j(S,\sigma)$, and R does not include replication and parallel operators, $R \xrightarrow{\lambda} R'$ is formally defined, coherently with the applied π-calculus semantics, by the rules in Table 7.

If instead $R = j(S,\sigma)$ for some S and σ, the meaning given to $R \xrightarrow{c!M} R'$ is that $j(S,\sigma)$ can output a message symbolically represented by M on a channel symbolically represented by c and evolve into R'. If S is a Java implementation statement generated by the $J()$ function, the only possibility for S to output a message is that S takes the form $id_0.send(id_1);S'$. Hence, also considering all the other cases, the possible evolutions of $R = j(S,\sigma)$ take the form $j(S,\sigma) \xrightarrow{\lambda} j(S',\sigma')$. These evolutions are formally defined by the rules in Table 8.

The formal semantics of a refined model can now be represented by means of an LTS whose set of states Σ is the set of configurations $(\mathcal{E},\mathcal{R})$, where $\mathcal{E}$ is a set of symbolic names such that $\mathcal{E}$ includes all the free symbolic names occurring in the extended applied π-calculus processes that belong to $\mathcal{R}$.

The evolutions of these configurations are represented by a labeled transition relation where the set of labels includes the label $\lambda = c,M$ which is used for transitions corresponding to the transmission of symbolic message M on channel c, $\lambda = \nu M$, which is used for transitions corresponding to the creation of fresh data symbolically represented by M, $\lambda = event(n,M)$, which is used for transitions corresponding to the generation of event $n(M)$ in one of the processes, and $\lambda = \tau$, which is used for transitions corresponding to internal, unobservable events.

The rules that define the transition relation are specified in Table 9.

Finally, the initial state of the LTS that represents a refined model M_r is the configuration $(fn(PV(M_r)), \{PV(M_r)\})$, where $PV(M_r) = PV(DoRun(M_r))$, and $DoRun()$ is lifted to be applied to refined models too in this way: $DoRun(C_r,c^0) = DoRun(c^0)$, i.e. $DoRun(M_r)$ is the doRun method body of the initial (scenario) class of M_r. Here, it is also necessary to extend the definition of function $PV()$ to concrete implementation statements. Simply, if $stat$ is the body of the $doRun$ method of a concrete implementation class, $PV(stat) = j(stat',\emptyset)$ where $stat'$ is the statement in the try branch of the try-catch statement of a refined Java statement.

The LTS of a refined model M_r, as just defined, is denoted by $lts(M_r)$ while the traces of a refined model M_r are defined as the traces of its LTS, i.e. $traces(M_r) = traces(lts(M_r))$.

Table 9. Definition of the labeled reduction relation for refined model configurations

$(\mathcal{E}, \mathcal{R} \cup \{0\})$	$\xrightarrow{\tau}$	$(\mathcal{E}, \mathcal{R})$	
$(\mathcal{E}, \mathcal{R} \cup \{!R\})$	$\xrightarrow{\tau}$	$(\mathcal{E}, \mathcal{R} \cup \{R, !R\})$	
$(\mathcal{E}, \mathcal{R} \cup \{R_1 \mid R_2\})$	$\xrightarrow{\tau}$	$(\mathcal{E}, \mathcal{R} \cup \{R_1, R_2\})$	
$(\mathcal{E}, \mathcal{R} \cup \{R_1, R_2\})$	$\xrightarrow{c,M}$	$(\mathcal{E}, \mathcal{R} \cup \{R'_1, R'_2\})$	if $R_1 \xrightarrow{c!M} R'_1$ and $R_2 \xrightarrow{c?M} R'_2$
$(\mathcal{E}, \mathcal{R} \cup \{R\})$	$\xrightarrow{\nu M}$	$(\mathcal{E} \cup \{M\}, \mathcal{R} \cup \{R'[M/M']\})$	if $R \xrightarrow{\nu M'} R'$ and $M \notin \mathcal{E}$
$(\mathcal{E}, \mathcal{R} \cup \{R\})$	$\xrightarrow{event(n,M)}$	$(\mathcal{E}, \mathcal{R} \cup \{R'\})$	if $R \xrightarrow{event(n,M)} R'$
$(\mathcal{E}, \mathcal{R} \cup \{j(start(id_1, \cdots, id_n), \sigma)\})$	$\xrightarrow{\tau}$	$(\mathcal{E}, \mathcal{R} \cup \{!j(DoRun(id_1), \sigma) \mid \cdots \mid !j(DoRun(id_n), \sigma)\})$	
$(\mathcal{E}, \mathcal{R} \cup \{R\})$	$\xrightarrow{\tau}$	$(\mathcal{E}, \mathcal{R} \cup \{R'\})$	if $R \xrightarrow{\tau} R'$

8. Soundness Theorem

This section presents a proof of soundness for the proposed refinement approach. Intuitively, what we aim to prove is that if a property specified in the abstract model M_a has been proved by ProVerif on the $PV(M_a)$ applied π-calculus model, then the same property still holds when one (or more) of the protocol roles is substituted by the concrete Java program that implements that role.

Among the properties that ProVerif can prove, this paper considers correspondence properties [Bla09]. These properties are trace properties, i.e. they are defined for each single execution trace of an applied π-calculus model, and they are defined to hold on a model if and only if they hold on all its traces. Correspondence properties include secrecy and authentication properties, i.e. the properties that are most commonly used in security protocol verification.

For example, if $Init$ is a set of messages, any process having initial knowledge $Init$ is said to be an $Init$-attacker. Then, it is possible to define secrecy of a data term M against $Init$-attackers as follows: a protocol satisfies this property if no $Init$-attacker can get M when interacting with the protocol. This secrecy property can be formally defined as a trace property that must hold for all the traces of all the models that can be obtained by combining the protocol model with an $Init$-attacker. This trace property is true on a trace t if and only if, during t, the message M is never transmitted on a channel that belongs to $Init$ (in fact, if the attacker gets M, the attacker can also output M on a channel that belongs to $Init$, hence the absence of a transmission of M on these channels implies that the attacker cannot get M).

Similarly, an authentication property is usually expressed as a correspondence between events that occur in the protocol actors. For example, if event $startSession(x)$ is executed by a protocol role when it gets engaged in a session of the protocol in which it agrees on some data x, and event $endSession(x)$ is executed by another actor when it correctly terminates a protocol session in which it has agreed on x, authentication can be expressed by requiring that, even in the presence of $Init$-attackers, for each $endSession(x)$ that occurs, there has been a corresponding $startSession(x)$ that occurred in the past. Once again, this property can be formally stated as a trace property that must hold for all the traces of all the models that can be obtained by combining the protocol model with an $Init$-attacker. Of course, in this case, the property is true for a trace if the occurrences of the startSession and endSession events within the trace fulfill the above correspondence.

In general, the correspondence properties that can be verified by ProVerif depend only on the ordered sequence of occurrence of the following two kinds of execution events:

- the transmission of a message M on a channel c, denoted by (c, M) in this paper.
- the occurrence of an event named n associated with data M, denoted by $event(n, M)$ in this paper.

The labeled reduction relation defined in Table 9 represents the evolution of a refined process (or of a fully abstract applied π-calculus process, which is a special case) with labels that represent the occurrence of message transmissions and of events, as well as the creation of fresh data. In this way, the sequence of labels of an execution trace is a sort of digest that incorporates enough information to establish if that trace satisfies the properties to be verified. Indeed, for technical reasons, the labels include more than what is strictly necessary for determining the truth of the security properties that are being considered.

The truth of a correspondence property on a trace $t = \lambda_1, \lambda_2, \cdots$ can be expressed by a trace pred-

icate $\phi(t)$. For example, the secrecy of M against $Init$-attackers can be expressed as the trace predicate $secrecy_{M,Init}(t)$, which is true for trace $t = \lambda_1, \lambda_2, \cdots$ if $\lambda_i \neq c, M$ for any $c \in Init$.

As the same type of labels is used both for the semantics of abstract models and for the semantics of refined models, the possible evolutions of an abstract model and of a refined model can be both represented by LTSs having labels in the same alphabet. As already observed, these systems are enough for establishing the truth of the security properties considered in this paper, both on abstract models and on refined models. Given a trace property which is a trace predicate ϕ, and a refined model M_r, $M_r \models \phi$ means that ϕ holds for all the traces of M_r, i.e. $M_r \models \phi \iff \forall t \in traces(M_r).\phi(t)$.

The soundness theorem can then be formulated as follows.

Theorem 8.1. For each abstract model M_a, refinement r, and trace property ϕ, $M_a \models \phi \implies r(M_a) \models \phi$.

The proof of Theorem 8.1 can be developed by showing that $lts(r(M_a))$ weakly simulates $lts(M_a)$. This is expressed by the following lemma.

Lemma 8.2. For each abstract model M_a, and refinement r, $lts(r(M_a))$ weakly simulates $lts(M_a)$.

In order to be able to prove Lemma 8.2, let us introduce the definition of a relation $\mathcal{S} \subseteq \Sigma_1 \times \Sigma_2$, where Σ_1 is a set of concrete states and Σ_2 is a set of abstract states, that we will claim to be a weak simulation relation relating the initial states of $lts(r(M_a))$ and $lts(M_a)$.

Definition 6 (Relation $\mathcal{S}$). $((\mathcal{E}_1, \mathcal{R}_1), (\mathcal{E}_2, \mathcal{R}_2)) \in \mathcal{S}$ iff $\mathcal{E}_1 = \mathcal{E}_2$ and there exists a bijection $b_p \subseteq \mathcal{R}_1 \times \mathcal{R}_2$ such that $(R_1, R_2) \in b_p$ implies $R_1 = R_2$ or $R_1 = j(S_1, \sigma)$ for some S_1 and σ such that

$$S_1 = THR \vee S_1 = return \vee \exists S_2.((S_1 = J(S_2)) \wedge (R_2 = PV(S_2)[PV(\sigma)])). \tag{2}$$

Lemma 8.2 will be proved by using the following additional lemmas.

Lemma 8.3. $\forall((\mathcal{E}, \mathcal{R}_1), (\mathcal{E}, \mathcal{R}_2)) \in \mathcal{S}.\ \forall(\mathcal{E}', \mathcal{R}_1').$

$$(\mathcal{E}, \mathcal{R}_1) \xrightarrow{\tau} (\mathcal{E}', \mathcal{R}_1') \implies \exists \mathcal{R}_2'.((\mathcal{E}, \mathcal{R}_2) \stackrel{\tau}{\Rightarrow} (\mathcal{E}', \mathcal{R}_2') \ \wedge \ ((\mathcal{E}', \mathcal{R}_1'), (\mathcal{E}', \mathcal{R}_2')) \in \mathcal{S}). \tag{3}$$

Lemma 8.4. $\forall((\mathcal{E}, \mathcal{R}_1), (\mathcal{E}, \mathcal{R}_2)) \in \mathcal{S}.\ \forall(\mathcal{E}', \mathcal{R}_1').\ \forall \lambda \in \Lambda - \{\tau\}.$

$$(\mathcal{E}, \mathcal{R}_1) \xrightarrow{\lambda} (\mathcal{E}', \mathcal{R}_1') \implies \exists \mathcal{R}_2'.((\mathcal{E}, \mathcal{R}_2) \stackrel{\lambda}{\rightarrow} (\mathcal{E}', \mathcal{R}_2') \ \wedge \ ((\mathcal{E}', \mathcal{R}_1'), (\mathcal{E}', \mathcal{R}_2')) \in \mathcal{S}). \tag{4}$$

The proofs of Lemmas 8.2, 8.3, 8.4 will be given after the proof of Theorem 8.1.

Proof of Theorem 8.1 We prove the equivalent statement $r(M_a) \not\models \phi \implies M_a \not\models \phi$. By definition, $r(M_a) \not\models \phi$ implies there exists at least one trace $t \in traces(r(M_a))$ such that $\phi(t)$ is false. By Lemma 8.2 $lts(r(M_a))$ weakly simulates $lts(M_a)$, which is known to imply that $otraces(lts(r(M_a))) \subseteq otraces(lts(M_a))$ ([GV15], section 2.4.2). As a consequence, $t\backslash\{\tau\} \in traces(lts(M_a))$, which implies that there exists a trace $t' \in traces(M_a)$ such that $t'\backslash\{\tau\} = t\backslash\{\tau\}$. As ϕ is a trace property, ϕ depends on observable events only. Hence, as $\phi(t)$ is false, $\phi(t\backslash\{\tau\})$ is false too, and so is $\phi(t')$, which implies $M_a \not\models \phi$.

□

Proof of Lemma 8.2 By Lemmas 8.3 and 8.4, and by definition 4, and by considering that $\forall s, s', \lambda.\ s \xrightarrow{\lambda} s' \implies s \stackrel{\lambda}{\Rightarrow} s'$, it follows that $\mathcal{S}$, as defined in definition 6, is a weak simulation relation.

In order to prove the lemma, we need to show that the initial states of $lts(r(M_a))$ and $lts(M_a)$ are related by $\mathcal{S}$. According to the definition, each initial state is a configuration that includes a single (extended) applied π-calculus process. More precisely, if c^0 is the initial class of M_a, the initial state of $lts(M_a)$ is $(fn(PV(DoRun(c^0))), \{PV(DoRun(c^0))\})$. As the initial class of a model is not refined, c^0 is also the initial class of $r(M_a)$, and the initial state of $lts(r(M_a))$ is $(fn(PV(DoRun(c^0))), \{j(DoRun(c^0), \emptyset)\})$. These two states are related by $\mathcal{S}$ because the first component of each configuration (i.e. the set of free names) is the same and there exists a single bijection that maps the single process of each configuration onto the single process of the other configuration. This bijection satisfies (2) because $R_1 = j(stat, \emptyset)$, and $J(stat) = stat$, and $R_2 = PV(stat)[PV(\emptyset)] = PV(stat)$.

□

Proof of Lemma 8.3 As $((\mathcal{E},\mathcal{R}_1),(\mathcal{E},\mathcal{R}_2))\in\mathcal{S}$, from the definition of $\mathcal{S}$ we have that there exists a $b_p \subseteq \mathcal{R}_1 \times \mathcal{R}_2$ such that $(R_1,R_2) \in b_p$ implies $R_1 = R_2$ or $R_1 = j(S_1,\sigma)$ for some S_1 and σ such that (2) holds.

From the left hand side of implication (3) and from Table 9, it follows that $\mathcal{R}_1$ must take the form $\mathcal{R}_1 = \mathcal{R} \cup \{R_1\}$, where R_1 either takes one of the forms 0, $!R$, $R_a|R_b$, $j(start(id_1,\cdots,id_n),\sigma)$, or it is such that $R_1 \xrightarrow{\tau} R_1'$ for some R_1'. In fact, no other cases are possible, according to Table 9. Moreover, again by inspection of Table 9, we have that $\mathcal{E}' = \mathcal{E}$ and $\mathcal{R}_1' \supseteq \mathcal{R}$.

Let us denote by $\mathcal{Q}$ the image of $\mathcal{R}$ according to b_p. We have that $\mathcal{R}_2 = \mathcal{Q} \cup \{b_p(R_1)\}$. Now we distinguish two possible cases.

Case 1 If $b_p(R_1) = R_1$, then $\mathcal{R}_2 = \mathcal{Q} \cup \{R_1\}$ and, from Table 9, $(\mathcal{E},\mathcal{R}_2) \xrightarrow{\tau} (\mathcal{E},\mathcal{R}_2')$ for some $\mathcal{R}_2' \supseteq \mathcal{Q}$, which proves that $\exists \mathcal{R}_2'.((\mathcal{E},\mathcal{R}_2) \stackrel{\tau}{\Rightarrow} (\mathcal{E}',\mathcal{R}_2'))$.

We still have to prove that, in this case, $((\mathcal{E}',\mathcal{R}_1'),(\mathcal{E}',\mathcal{R}_2')) \in \mathcal{S}$, i.e., by definition of $\mathcal{S}$, we have to prove the existence of a bijection $b_p' \subseteq \mathcal{R}_1' \times \mathcal{R}_2'$ that matches the definition of $\mathcal{S}$. We can define b_p' as follows: $\forall R \in \mathcal{R}.\ (R, b_p(R)) \in b_p'$, and $\forall R \in \mathcal{R}_1' - \mathcal{R}.\ (R,R) \in b_p'$. Since b_p is a bijection by assumption, b_p' is a bijection too. This definition of b_p' trivially matches the definition of $\mathcal{S}$.

Case 2 If instead $b_p(R_1) \neq R_1$, based on the definition of $\mathcal{S}$, R_1 takes the form $j(S_1,\sigma)$, and (2) holds. In this case, S_1 cannot take the form $start(id_1,\cdots,id_n)$ because the start statement cannot occur in the Java code of protocol roles. Hence, the only rule in Table 9 that applies is the last one, which implies that R_1' exists such that $R_1 \xrightarrow{\tau} R_1'$. Since $R_1 = j(S_1,\sigma)$, from the evolution rules of processes that take this form (Table 8) we have that $R_1' = j(S_1',\sigma')$ for some S_1' and σ' such that $(S_1,\sigma) \to (S_1',\sigma')$ according to the JavaSPI semantics (Table 6), and $\forall \lambda \neq \tau.\ j(S_1,\sigma) \stackrel{\lambda}{\not\to} j(S_1',\sigma')$.

Let us consider now the different cases that correspond to the different rules of Table 6 that can determine the evolutions $(S_1,\sigma) \to (S_1',\sigma')$.

Rule 1 S_1 is final *typeC id* = new $typeC(arg\text{-}list_c)$; *stat*, and $type = \alpha(typeC)$ is not a Name, and $S_1' = stat$ and $\sigma' = \sigma \cup \{(id, type(\sigma(arg\text{-}list_a)))\}$ with $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id)$.
In this case, as we know that $\exists S_2.S_1 = J(S_2)$, based on Table 4, we have that S_2 is

final *type id* = new $type(arg\text{-}list_2)$; $stat_2$

with $stat_2$ such that $stat = J(stat_2)$ and $arg\text{-}list_2$ such that $arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id)$.
Moreover, $R_2 = PV(S_2)[PV(\sigma)] = (\text{let } PV(id) = PV(type)(PV(arg\text{-}list_2)) \text{ in } PV(stat_2))[PV(\sigma)]$.
Based on Table 5 and Table 7, we have that $R_2 \xrightarrow{\tau} R_2'$ with

$$R_2' = PV(stat_2)[PV(type)(PV(arg\text{-}list_2))/PV(id)][PV(\sigma)]$$

According to Table 9, this implies that $(\mathcal{E},\mathcal{R}_2) \xrightarrow{\tau} (\mathcal{E},\mathcal{R}_2')$ with

$$\mathcal{R}_2' = \mathcal{Q} \cup \{PV(stat_2)[PV(type)(PV(arg\text{-}list_2))/PV(id)][PV(\sigma)]\}$$

We still have to prove that $((\mathcal{E},\mathcal{R}_1'),(\mathcal{E},\mathcal{R}_2')) \in \mathcal{S}$. In order to prove it, we define a bijection $b_p' \subseteq \mathcal{R}_1' \times \mathcal{R}_2'$ in this way:

$$\forall R \in \mathcal{R}.b_p'(R) = b_p(R) \text{ and } b_p'(j(stat,\sigma')) = PV(stat_2)[PV(type)(PV(arg\text{-}list_2))/PV(id)][PV(\sigma)].$$

As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b_p' matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(stat,\sigma'), b_p'(j(stat,\sigma'))$, i.e. we have to show that (2) holds with the substitutions $S_1 \to stat$, $\sigma \to \sigma'$, and $R_2 \to R_2'$. Let us apply the above substitutions in (2), along with equation $\sigma' = \sigma \cup \{(id, type(\sigma(arg\text{-}list_a)))\}$.
Moreover, since $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id)$ and $arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id)$, we have finally that $arg\text{-}list_a = AbsArg(ConcrArg(arg\text{-}list_2, id), id) = arg\text{-}list_2$.
If we choose $S_2 = stat_2$, we see that (2) with the above substitutions holds.

Rule 2 Only the case where PVarDef(id)≠PRIVATE is possible, because if PVarDef(id)=PRIVATE then $j(S_1,\sigma) \stackrel{\nu M}{\to} j(S_1',\sigma')$ for some M, which implies $j(S_1,\sigma) \stackrel{\tau}{\not\to} j(S_1',\sigma')$.
In this case S_1 is final *typeC id* = new $typeC(arg\text{-}list_c)$; *stat*, $\alpha(typeC)$ is a Name, $S_1' = stat$ and $\sigma' = \sigma \cup \{(id, M)\}$ with M such that $M \notin gnd(Im(\sigma))$.
In this case, as we know that $\exists S_2.S_1 = J(S_2)$, based on Table 4, we have that S_2 is

final *type id* = new $type(arg\text{-}list_2)$; $stat_2$

with $stat_2$ such that $stat = J(stat_2)$, $type = \alpha(typeC)$, and $arg\text{-}list_2$ such that

$$arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id).$$

Moreover, $R_2 = PV(S_2)[PV(\sigma)] = (\text{let } PV(id) = PV(type)(PV(arg\text{-}list_2)) \text{ in } PV(stat_2))[PV(\sigma)]$.
Based on Table 5 and Table 7, we have that $R_2 \xrightarrow{\tau} R_2'$ with

$$R_2' = PV(stat_2)[PV(type)(PV(arg\text{-}list_2))/PV(id)][PV(\sigma)]$$

According to Table 9, this implies that $(\mathcal{E}, \mathcal{R}_2) \xrightarrow{\tau} (\mathcal{E}, \mathcal{R}_2')$ with

$$\mathcal{R}_2' = \mathcal{Q} \cup \{PV(stat_2)[PV(type)(PV(arg\text{-}list_2))/PV(id)][PV(\sigma)]\}$$

We still have to prove that $((\mathcal{E}, \mathcal{R}_1'), (\mathcal{E}, \mathcal{R}_2')) \in \mathcal{S}$. In order to prove it, we define a bijection $b_p' \subseteq \mathcal{R}_1' \times \mathcal{R}_2'$ in this way:

$$\forall R \in \mathcal{R}. b_p'(R) = b_p(R) \text{ and } b_p'(j(stat, \sigma')) = PV(stat_2)[PV(type)(PV(arg\text{-}list_2))/PV(id)][PV(\sigma)].$$

As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b_p' matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(stat, \sigma'), b_p'(j(stat, \sigma'))$, i.e. we have to show that (2) holds with the substitutions $S_1 \to stat$, $\sigma \to \sigma'$, and $R_2 \to R_2'$. Let us apply the above substitutions in (2), along with equation $\sigma' = \sigma \cup \{(id, type(\sigma(arg\text{-}list_a)))\}$.
Moreover, since $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id)$ and $arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id)$, we have finally that $arg\text{-}list_a = AbsArg(ConcrArg(arg\text{-}list_2, id), id) = arg\text{-}list_2$.
If we choose $S_2 = stat_2$, we see that (2) with the above substitutions holds.

Rule 3 This case is not possible because $j(S_1, \sigma) \xrightarrow{c?M} j(S_1', \sigma')$ for some c and M, which implies $j(S_1, \sigma) \xrightarrow{\tau} \not\to j(S_1', \sigma')$.

Rule 4 S_1 is final $typeC\ id_1 = id_0.id_2(arg\text{-}list_c); stat$, and $\text{PFun}(id_0, id_2)(\sigma(L(id_0, arg\text{-}list_a))) \to M$ with $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id_0, id_2)$, and $S_1' = stat$ and $\sigma' = \sigma \cup \{(id_1, M)\}$.
In this case, as we know that $\exists S_2. S_1 = J(S_2)$, based on Table 4, we have that S_2 is

$$\text{final } type\ id_1 = id_0.id_2(arg\text{-}list_2); stat_2$$

with $stat_2$ such that $stat = J(stat_2)$ and $arg\text{-}list_2$ such that $arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id_0, id_2)$.
Moreover, $R_2 = PV(S_2)[PV(\sigma)]$ evaluates to
(let x suchthat member: x, true_false_set in
let $PV(id_1) = PFun(id_0, id_2)(L(PV(id_0), PV(arg\text{-}list_2), x))$ in
$PV(stat))[PV(\sigma)]$
Since $arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id_0, id_2)$, we have that

$$AbsArg(arg\text{-}list_c, id_0, id_2) = AbsArg(ConcrArg(arg\text{-}list_2, id_0, id_2), id_0, id_2) = arg\text{-}list_2$$

which implies that $\text{PFun}(id_0, id_2)(\sigma(L(id_0, arg\text{-}list_2))) \to M$, based on the initial assumptions coming from rule 8. In turn, this implies that $\text{PFun}(id_0, id_2)(\sigma(L(id_0, arg\text{-}list_2, false))) \to M$.
Based on Table 5 and Table 7, we have that $R_2 \xrightarrow{\tau}\xrightarrow{\tau} R_2'$ with $R_2' = PV(stat)[false/x][M/PV(id_1)][PV(\sigma)]$.
According to Table 9, this implies that $(\mathcal{E}, \mathcal{R}_2) \xrightarrow{\tau}\xrightarrow{\tau} (\mathcal{E}, \mathcal{R}_2')$ with $\mathcal{R}_2' = \mathcal{Q} \cup \{R_2'\}$.
We still have to prove that $((\mathcal{E}, \mathcal{R}_1'), (\mathcal{E}, \mathcal{R}_2')) \in \mathcal{S}$. In order to prove it, we define a bijection $b_p' \subseteq \mathcal{R}_1' \times \mathcal{R}_2'$ in this way: $\forall R \in \mathcal{R}. b_p'(R) = b_p(R)$ and $b_p'(j(stat, \sigma')) = PV(stat)[false/x][M/PV(id_1)][PV(\sigma)]$. As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b_p' matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(stat, \sigma'), b_p'(j(stat, \sigma'))$, i.e. we have to show that (2) holds with the following substitutions

$$S_1 \to stat,\ \sigma \to \sigma',\ R_2 \to PV(stat)[false/x][M/PV(id_1)][PV(\sigma)].$$

Let us apply the above substitutions in (2), along with equation $\sigma' = \sigma' = \sigma \cup \{(id_1, M)\}$. If we choose $S_2 = stat$, we see that (2) with the above substitutions holds.

Rule 5 S_1 is final $typeC\ id_1 = id_0; stat$, and $S_1' = stat$, and $\sigma' = \sigma \cup \{(id_1, \sigma(id_0))\}$.
In this case, as we know that $\exists S_2. S_1 = J(S_2)$, based on Table 4, we have that $S_2 = \text{final } typeC\ id_1 = id_0; stat_2$ with $stat_2$ such that $stat = J(stat_2)$.

Moreover, $R_2 = PV(S_2)[PV(\sigma)] = (\text{let } PV(id_1) = PV(id_0) \text{ in } PV(stat_2))[PV(\sigma)] = \text{let } \sigma(id_1) = \sigma(id_0)$ in $PV(stat_2)[PV(\sigma)]$.
Based on Table 5 and Table 7, we have that $R_2 \xrightarrow{\tau} R_2'$ with $R_2' = PV(stat_2)[PV(\sigma)][\sigma(id_0)/\sigma(id_1)]$.
From this result and from Table 9, we have finally that $(\mathcal{E}, \mathcal{R}_2) \xrightarrow{\tau} (\mathcal{E}, \mathcal{R}_2')$ with $\mathcal{R}_2' = \mathcal{Q} \cup \{R_2'\}$.
We still have to prove that $((\mathcal{E}, \mathcal{R}_1'), (\mathcal{E}, \mathcal{R}_2')) \in \mathcal{S}$.
In order to prove it, we define a bijection $b_p' \subseteq \mathcal{R}_1' \times \mathcal{R}_2'$ in this way: $\forall R \in \mathcal{R}.b_p'(R) = b_p(R)$ and $b_p'(j(stat_1, \sigma)) = PV(stat_2)[PV(\sigma)]$. As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b_p' matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(stat_1, \sigma), b_p'(j(stat_1, \sigma))$, i.e. we have to show that (2) holds with the substitutions $S_1 \rightarrow stat$, and $R_2 \rightarrow R_2'$. Let us apply the above substitutions in (2). The predicate holds because if we choose $S_2 = stat_2$, we have that $(S_1 = J(S_2))$ and $(R_2 = PV(S_2)[PV(\sigma)])$.

Rule 6 This case is not possible because $j(S_1, \sigma) \overset{c!M}{\rightarrow} j(S_1', \sigma')$ for some c and M, which implies $j(S_1, \sigma) \overset{\tau}{\not\rightarrow} j(S_1', \sigma')$.

Rule 7 This case is not possible because $j(S_1, \sigma) \overset{event(n,M)}{\rightarrow} j(S_1', \sigma')$ for some M, which implies $j(S_1, \sigma) \overset{\tau}{\not\rightarrow} j(S_1', \sigma')$.

Rule 8 S_1 is $\text{if}(id_0.id_1(arg\text{-}list_c))\{stat_1\}\text{else}\{stat_2\}$, and $\text{PFun}(id_0, id_1)(\sigma(L(id_0, arg\text{-}list_a))) \rightarrow true$, with $arg\text{-}list_a = AbsArg(arg\text{-}list_c, id_0, id_1)$, and $S_1' = stat_1$, and $\sigma' = \sigma$.
In this case, as we know that $\exists S_2.S_1 = J(S_2)$, based on Table 4, we have that S_2 is

$$\text{if}(cexpr_2)\{stat_{21}\}\text{else}\{stat_{22}\}$$

with $J(cexpr_2) = id_0.id_1(arg\text{-}list_c)$, $J(stat_{21}) = stat_1$ and $J(stat_{22}) = stat_2$. Again, based on Table 4, we have that $cexpr_2 = id_0.id_1(arg\text{-}list_2)$ with $arg\text{-}list_2$ such that $arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id_0, id_1)$.
Moreover, $R_2 = PV(S_2)[PV(\sigma)]$, which evaluates to

$$(\text{if}(PV(cexpr_2)))\{PV(stat_{21})\}\text{else}\{PV(stat_{22})\})[PV(\sigma)] =$$
$$\text{if}(PV(cexpr_2)[PV(\sigma)]))\{PV(stat_{21})[PV(\sigma)]\}\text{else}\{PV(stat_{22})[PV(\sigma)]\}$$

Based on Table 3, we have that $PV(cexpr_2)$ is $\text{PFun}(id_0, id_1)(L(PV(id_0), PV(arg\text{-}list_2))) = true$ and $PV(cexpr_2)[PV(\sigma)]$ is $\text{PFun}(id_0, id_1)(\sigma(L(id_0, arg\text{-}list_2))) = true$.
Since $arg\text{-}list_c = ConcrArg(arg\text{-}list_2, id_0, id_1)$, we have that

$$AbsArg(arg\text{-}list_c, id_0, id_1) = AbsArg(ConcrArg(arg\text{-}list_2, id_0, id_1), id_0, id_1) = arg\text{-}list_2$$

which implies that $\text{PFun}(id_0, id_1)(\sigma(L(id_0, arg\text{-}list_2))) \rightarrow true$, based on the initial assumptions coming from rule 8.
Hence, we have that $PV(cexpr_2)[PV(\sigma)]$ reduces to $true = true$ and, based on Table 5 and Table 7, we have that $R_2 \xrightarrow{\tau} R_2'$ with $R_2' = PV(stat_{21})[PV(\sigma)]$.
From this result and from Table 9, we have finally that $(\mathcal{E}, \mathcal{R}_2) \xrightarrow{\tau} (\mathcal{E}, \mathcal{R}_2')$ with $\mathcal{R}_2' = \mathcal{Q} \cup \{R_2'\}$
We still have to prove that $((\mathcal{E}, \mathcal{R}_1'), (\mathcal{E}, \mathcal{R}_2')) \in \mathcal{S}$.
In order to prove it, we define a bijection $b_p' \subseteq \mathcal{R}_1' \times \mathcal{R}_2'$ in this way: $\forall R \in \mathcal{R}.b_p'(R) = b_p(R)$ and $b_p'(j(stat_1, \sigma)) = PV(stat_{21})[PV(\sigma)]$. As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b_p' matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(stat_1, \sigma), b_p'(j(stat_1, \sigma))$, i.e. we have to show that (2) holds with the substitutions $S_1 \rightarrow stat_1$, and $R_2 \rightarrow PV(stat_{21})[PV(\sigma)]$. Let us apply the above substitutions in (2). The predicate holds because if we choose $S_2 = stat_{21}$, we see that $(S_1 = J(S_2))$ and $(R_2 = PV(S_2)[PV(\sigma)])$.

Rule 9 The proof is similar to the one for Rule 8. It is omitted here for brevity.

Rule 10 $S_1 = stat$, and $S_1' = THR$ and $\sigma' = \sigma$.
In this case, we claim that $((\mathcal{E}, \mathcal{R}_1'), (\mathcal{E}, \mathcal{R}_2)) \in \mathcal{S}$, which would prove the lemma in this case, since trivially $(\mathcal{E}, \mathcal{R}_2) \overset{\tau}{\Rightarrow} (\mathcal{E}, \mathcal{R}_2)$. Our claim can be proved by defining a bijection $b_p' \subseteq \mathcal{R}_1' \times \mathcal{R}_2$ in this way: $\forall R \in \mathcal{R}.b_p'(R) = b_p(R)$ and $b_p'(j(THR, \sigma)) = b_p(j(S_1, \sigma))$. As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b_p' matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(THR, \sigma), b_p(j(S_1, \sigma))$, which is true because the first term of (2) applies.

As the proof has been completed for all the rules of Table 6, the lemma is proved.

□

Proof of Lemma 8.4 The proof steps are similar to the ones of the previous lemma.

Let us prove the lemma separately for the different forms of λ.

Case 1: $\lambda = event(n, M)$

From our first assumption and from the definition of $\mathcal{S}$, we have that there exists a $b_p \subseteq \mathcal{R}_1 \times \mathcal{R}_2$ such that $(R_1, R_2) \in b_p$ implies $R_1 = R_2$ or $R_1 = j(S_1, \sigma)$ for some S_1 and σ such that (2) holds.

From our second assumption and from Table 9, $\mathcal{R}_1$ must take the form $\mathcal{R}_1 = \mathcal{R} \cup \{R_1\}$, with R_1 such that $R_1 \overset{event(n,M)}{\rightarrow} R'_1$ for some R'_1.

Based on Table 7, we have that $R_1 = event\ n(M); R'_1$. Moreover, $\mathcal{E}' = \mathcal{E}$ and $\mathcal{R}'_1 = \mathcal{R} \cup \{R'_1\}$.

Let us denote by $\mathcal{Q}$ the image of $\mathcal{R}$ according to b_p. We have that $\mathcal{R}_2 = \mathcal{Q} \cup \{b_p(R_1)\}$. Now we distinguish two possible cases.

If $b_p(R_1) = R_1$, then $\mathcal{R}_2 = \mathcal{Q} \cup \{R_1\}$ and, from Table 9, $(\mathcal{E}, \mathcal{R}_2) \overset{event(n,M)}{\rightarrow} (\mathcal{E}, \mathcal{R}'_2)$ with $\mathcal{R}'_2 = \mathcal{Q} \cup \{R'_1\}$, which proves that $\exists \mathcal{R}'_2.(\mathcal{E}, \mathcal{R}_2) \overset{\lambda}{\rightarrow} (\mathcal{E}', \mathcal{R}'_2)$.

In order to also prove that $((\mathcal{E}, \mathcal{R}'_1), (\mathcal{E}, \mathcal{R}'_2)) \in \mathcal{S}$, we have to prove the existence of a bijection $b'_p \subseteq \mathcal{R}'_1 \times \mathcal{R}'_2$ that matches the definition of $\mathcal{S}$. We can define b'_p as follows: $\forall R \in \mathcal{R}.\ (R, b_p(R)) \in b'_p$, and $\forall R \in \mathcal{R}'_1 - \mathcal{R}.\ (R, R) \in b'_p$. This definition trivially matches the definition of $\mathcal{S}$.

If instead $b_p(R_1) \neq R_1$, based on the definition of $\mathcal{S}$, R_1 takes the form $j(S_1, \sigma)$, and (2) holds. In this case, from Table 9, we have that R'_1 exists such that $R_1 \overset{event(n,M)}{\rightarrow} R'_1$. Since $R_1 = j(S_1, \sigma)$, from the evolution rules of processes that take this form (Table 8) we have that S_1 takes the form $event(ev\text{-}arg\text{-}list); S'_1$ for some *ev-arg-list* such that $\sigma(ev\text{-}arg\text{-}list) = L(n, M)$, and $R'_1 = j(S'_1, \sigma)$.

As in this case we know that $\exists S_2.S_1 = J(S_2)$, based on Table 4, we have that S_2 is $event(ev\text{-}arg\text{-}list); S'_2$ with S'_2 such that $J(S'_2) = S'_1$.

In this case, $\mathcal{R}_2 = \mathcal{Q} \cup \{R_2\}$ with $R_2 = PV(S_2)[PV(\sigma)]$ which, based on Table 3, is

$$(\text{event } ev\text{-}name(PV(arg\text{-}list)); PV(S'_2))[PV(\sigma)]$$

where $ev\text{-}arg\text{-}list = ev\text{-}name, arg\text{-}list$. Considering that $\sigma(ev\text{-}arg\text{-}list) = L(n, M)$, we have finally that

$$R_2 = \text{event } n(M); PV(S'_2)[PV(\sigma)]$$

This implies that, according to Table 7, $R_2 \overset{event(n,M)}{\rightarrow} R'_2$ with $R'_2 = PV(S'_2)[PV(\sigma)]$.

Hence, from Table 9 $(\mathcal{E}, \mathcal{R}_2) \overset{event(n,M)}{\rightarrow} (\mathcal{E}, \mathcal{R}'_2)$ with $\mathcal{R}'_2 = \mathcal{Q} \cup \{R'_2\}$, which proves that $\exists \mathcal{R}'_2.(\mathcal{E}, \mathcal{R}_2) \overset{\lambda}{\rightarrow} (\mathcal{E}', \mathcal{R}'_2)$.

In order to also prove that $((\mathcal{E}, \mathcal{R}'_1), (\mathcal{E}, \mathcal{R}'_2)) \in \mathcal{S}$, we have to prove the existence of a bijection $b'_p \subseteq \mathcal{R}'_1 \times \mathcal{R}'_2$ that matches the definition of $\mathcal{S}$. We can define b'_p as follows: $\forall R \in \mathcal{R}.\ (R, b_p(R)) \in b'_p$, and $b'_p(j(S'_1, \sigma)) = PV(S'_2)[PV(\sigma)]$. As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b'_p matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(S'_1, \sigma), b'_p(j(S'_1, \sigma))$, i.e. we have to show that (2) holds with the substitutions $S_1 \rightarrow S'_1$, and $R_2 \rightarrow R'_2$. Let us apply the above substitutions in (2). The predicate holds because if we choose $S_2 = S'_2$, we have that $(S_1 = J(S_2))$ and $(R_2 = PV(S_2)[PV(\sigma)])$.

Case 2: $\lambda = \nu(M)$

From our first assumption and from the definition of $\mathcal{S}$, we have that there exists a $b_p \subseteq \mathcal{R}_1 \times \mathcal{R}_2$ such that $(R_1, R_2) \in b_p$ implies $R_1 = R_2$ or $R_1 = j(S_1, \sigma)$ for some S_1 and σ such that (2) holds.

From our second assumption and from Table 9, $\mathcal{R}_1$ must take the form $\mathcal{R}_1 = \mathcal{R} \cup \{R_1\}$, with R_1 such that $R_1 \overset{\nu M}{\rightarrow} R'_1$ for some R'_1.

Based on Table 7, we have that $R_1 = new\ x; R''_1$ and $R'_1 = R''_1[M/x]$. Moreover, $\mathcal{E}' = \mathcal{E} \cup \{M\}$ and $\mathcal{R}'_1 = \mathcal{R} \cup \{R'_1\}$.

Let us denote by $\mathcal{Q}$ the image of $\mathcal{R}$ according to b_p. We have that $\mathcal{R}_2 = \mathcal{Q} \cup \{b_p(R_1)\}$. Now we distinguish two possible cases.

If $b_p(R_1) = R_1$, then $\mathcal{R}_2 = \mathcal{Q} \cup \{R_1\}$ and, from Table 9, $(\mathcal{E}, \mathcal{R}_2) \overset{event(M)}{\rightarrow} (\mathcal{E}, \mathcal{R}'_2)$ with $\mathcal{R}'_2 = \mathcal{Q} \cup \{R'_1\}$, which proves that $\exists \mathcal{R}'_2.(\mathcal{E}, \mathcal{R}_2) \overset{\lambda}{\rightarrow} (\mathcal{E}', \mathcal{R}'_2)$.

In order to also prove that $((\mathcal{E}, \mathcal{R}'_1), (\mathcal{E}, \mathcal{R}'_2)) \in \mathcal{S}$, we have to prove the existence of a bijection $b'_p \subseteq$

$\mathcal{R}'_1 \times \mathcal{R}'_2$ that matches the definition of $\mathcal{S}$. We can define b'_p as follows: $\forall R \in \mathcal{R}.\ (R, b_p(R)) \in b'_p$, and $\forall R \in \mathcal{R}'_1 - \mathcal{R}.\ (R, R) \in b'_p$. This definition trivially matches the definition of $\mathcal{S}$.

If instead $b_p(R_1) \neq R_1$, based on the definition of $\mathcal{S}$, R_1 takes the form $j(S_1, \sigma)$, and (2) holds.

In this case, from Table 9, we have that R'_1 exists such that $R_1 \stackrel{\nu M}{\rightarrow} R'_1$. Since $R_1 = j(S_1, \sigma)$, from the evolution rules of processes that take this form (Table 8) we have that S_1 takes the form final *typeC* id = new *typeC*(*arg-list*$_c$); S'_1, where *typeC* is a *Name* and PVarDef(id)=PRIVATE, and $R'_1 = j(S'_1, \sigma')$ with $\sigma' = \sigma \cup \{(id, M)\}$.

As in this case we know that $\exists S_2.S_1 = J(S_2)$, based on Table 4, we have that S_2 is final *type* id = new *type*(*arg-list*$_a$); S'_2 with S'_2 such that $J(S'_2) = S'_1$ and *arg-list*$_a$ such that $ConcrArg$(*arg-list*$_a, id$) = *arg-list*$_c$, and *type* = α(*typeC*).

In this case, $\mathcal{R}_2 = \mathcal{Q} \cup \{R_2\}$ with $R_2 = PV(S_2)[PV(\sigma)]$ which, based on Table 3, is

(new $PV(id); PV(S'_2))[PV(\sigma)]$

This implies that, according to Table 7, $R_2 \stackrel{\nu M}{\rightarrow} R'_2$ with $R'_2 = PV(S'_2)[PV(\sigma)][M/PV(id)]$. Hence, from Table 9, assuming $M \notin \mathcal{E}$, $(\mathcal{E}, \mathcal{R}_2) \stackrel{\nu M}{\rightarrow} (\mathcal{E} \cup \{M\}, \mathcal{R}'_2)$ with $\mathcal{R}'_2 = \mathcal{Q} \cup \{R'_2\}$, which proves that $\exists \mathcal{R}'_2.(\mathcal{E}, \mathcal{R}_2) \stackrel{\lambda}{\rightarrow} (\mathcal{E}', \mathcal{R}'_2)$.

In order to also prove that $((\mathcal{E} \cup \{M\}, \mathcal{R}'_1), (\mathcal{E} \cup \{M\}, \mathcal{R}'_2)) \in \mathcal{S}$, we have to prove the existence of a bijection $b'_p \subseteq \mathcal{R}'_1 \times \mathcal{R}'_2$ that matches the definition of $\mathcal{S}$. We can define b'_p as follows: $\forall R \in \mathcal{R}.\ (R, b_p(R)) \in b'_p$, and $b'_p(j(S'_1, \sigma)) = R'_2$. As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b'_p matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the pair $(j(S'_1, \sigma), b'_p(j(S'_1, \sigma))$, i.e. we have to show that (2) holds with the substitutions $S_1 \rightarrow S'_1$, $R_2 \rightarrow R'_2$, and $\sigma \rightarrow \sigma'$. Let us apply the above substitutions in (2). The predicate holds because if we choose $S_2 = S'_2$, we see that $(S_1 = J(S_2))$ and $(R_2 = PV(S_2)[PV(\sigma)])$.

Case 3: $\lambda = c, M$

From our first assumption and from the definition of $\mathcal{S}$, we have that there exists a $b_p \subseteq \mathcal{R}_1 \times \mathcal{R}_2$ such that $(R_1, R_2) \in b_p$ implies $R_1 = R_2$ or $R_1 = j(S_1, \sigma)$ for some S_1 and σ such that (2) holds.

From our second assumption and from Table 9, $\mathcal{R}_1$ must take the form $\mathcal{R}_1 = \mathcal{R} \cup \{R_{11}, R_{12}\}$, with R_{11} and R_{12} such that $R_{11} \stackrel{c!M}{\rightarrow} R'_{11}$ and $R_{12} \stackrel{c?M}{\rightarrow} R'_{12}$ for some R'_{11} and R'_{12}.

Based on Table 7, we have that $R_{11} = out(c, M); R'_{11}$ and $R_{12} = in(c, x); R''_{12}$ with $R'_{12} = R''_{12}[M/x]$. Moreover, $\mathcal{E}' = \mathcal{E}$ and $\mathcal{R}'_1 = \mathcal{R} \cup \{R'_{11}, R'_{12}\}$.

Let us denote by $\mathcal{Q}$ the image of $\mathcal{R}$ according to b_p. We have that $\mathcal{R}_2 = \mathcal{Q} \cup \{b_p(R_{11}), b_p(R_{12})\}$. There are different cases.

If $b_p(R_{11}) = R_{11}$, and $b_p(R_{12}) = R_{12}$ then $\mathcal{R}_2 = \mathcal{Q} \cup \{R_{11}, R_{12}\}$ and, from Table 9, $(\mathcal{E}, \mathcal{R}_2) \stackrel{c,M}{\rightarrow} (\mathcal{E}, \mathcal{R}'_2)$ with $\mathcal{R}'_2 = \mathcal{Q} \cup \{R'_{11}, R'_{12}\}$, which proves that $\exists \mathcal{R}'_2.(\mathcal{E}, \mathcal{R}_2) \stackrel{\lambda}{\rightarrow} (\mathcal{E}', \mathcal{R}'_2)$.

In order to also prove that $((\mathcal{E}, \mathcal{R}'_1), (\mathcal{E}, \mathcal{R}'_2)) \in \mathcal{S}$, we have to prove the existence of a bijection $b'_p \subseteq \mathcal{R}'_1 \times \mathcal{R}'_2$ that matches the definition of $\mathcal{S}$. We can define b'_p as follows: $\forall R \in \mathcal{R}.\ (R, b_p(R)) \in b'_p$, and $\forall R \in \mathcal{R}'_1 - \mathcal{R}.\ (R, R) \in b'_p$. This definition trivially matches the definition of $\mathcal{S}$.

If instead $b_p(R_{11}) \neq R_{11}$, based on the definition of $\mathcal{S}$, R_{11} takes the form $j(S_{11}, \sigma)$, and (2) holds. In this case, from Table 9, we have that R'_{11} exists such that $R_{11} \stackrel{c!M}{\rightarrow} R'_{11}$. Since $R_{11} = j(S_1, \sigma)$, from the evolution rules of processes that take this form (Table 8) we have that S_{11} takes the form $id_0.send(id_1); S'_{11}$, where $\sigma(id_0) = c$, and $\sigma(id_1) = M$, and $R'_{11} = j(S'_{11}, \sigma')$ with $\sigma' = \sigma$.

As in this case we know that $\exists S_{12}.S_{11} = J(S_{12})$, based on Table 4, we have that S_{12} is $id_0.send(id_1); S'_{12}$ with S'_{12} such that $J(S'_{12}) = S'_{11}$.

In this case, $\mathcal{R}_2 = \mathcal{Q} \cup \{R_{21}, R_{22}\}$ with $R_{21} = PV(S_{12})[PV(\sigma)]$ which, based on Table 3, is

$(out(PV(id_0), PV(id_1)); PV(S'_{12})[PV(\sigma)] = out(c, M); PV(S'_{12})[PV(\sigma)]$

This implies that, in this case, according to Table 7, $R_{21} \stackrel{c!M}{\rightarrow} R'_{21}$ with $R'_{21} = PV(S'_{12})[PV(\sigma)]$.

When $b_p(R_{12}) \neq R_{12}$ we get to a similar conclusion, i.e. $R_{22} \stackrel{c?M}{\rightarrow} R'_{22}$ with $R'_{22} = PV(S'_{22})[PV(\sigma)][M/x]$.

Hence, from Table 9, $(\mathcal{E}, \mathcal{R}_2) \stackrel{c,M}{\rightarrow} (\mathcal{E}, \mathcal{R}'_2)$ with $\mathcal{R}'_2 = \mathcal{Q} \cup \{R'_{21}, R'_{22}\}$, which proves that $\exists \mathcal{R}'_2.(\mathcal{E}, \mathcal{R}_2) \stackrel{\lambda}{\rightarrow} (\mathcal{E}', \mathcal{R}'_2)$.

In order to also prove that $((\mathcal{E}, \mathcal{R}'_1), (\mathcal{E}, \mathcal{R}'_2)) \in \mathcal{S}$, we have to prove the existence of a bijection $b'_p \subseteq$

$\mathcal{R}'_1 \times \mathcal{R}'_2$ that matches the definition of $\mathcal{S}$. We can define b'_p as follows: $\forall R \in \mathcal{R}.\ (R, b_p(R)) \in b'_p$, and $b'_p(j(S'_{11}, \sigma)) = R'_{21}$ and $b'_p(j(S'_{12}, \sigma)) = R'_{22}$.

As we already know that b_p matches the definition of $\mathcal{S}$, in order to prove that b'_p matches the definition of $\mathcal{S}$ too it is enough to show that this definition is matched by the remaining pairs $(j(S'_{11}, \sigma), b'_p(j(S'_{11}, \sigma))$, and $(j(S'_{12}, \sigma), b'_p(j(S'_{12}, \sigma))$.

In order to prove that this definition is matched by the first pair we have to show that (2) holds with the substitutions $S_1 \to S'_{11}$, and $R_2 \to R'_{21}$.

Let us apply the above substitutions in (2). The predicate holds because if we choose $S_2 = S'_{21}$, we see that $(S_1 = J(S_2))$ and $(R_2 = PV(S_2)[PV(\sigma)])$. The check in the other case is similar.

As the proof has been completed for all the forms of λ, the lemma is proved.

□

9. Security Guarantees and Necessary Assumptions

Having proved the soundness theorem, it is possible now to draw some conclusions about the security guarantees that the use of JavaSPI provides on the resulting protocol implementation code, and the assumptions under which these guarantees hold.

Let us assume that an abstract protocol model has been developed with JavaSPI, and that the corresponding applied π-calculus model has been generated, and that ProVerif has proved this model satisfies a certain trace property (e.g. a secrecy or authentication property). Now, let us assume an interoperable implementation of one of the protocol roles is developed by means of the JavaSPI refinement procedure (i.e. annotation of the class c^i modeling the role, code generation, and possibly manual writing of the necessary marshaling layers), and that a concrete resulting class $r(c^i)$ is finally obtained. By Theorem 8.1, the abstract model with c^i replaced by $r(c^i)$ still satisfies the same trace property, i.e. the refinement of the abstract model into an interoperable implementation has been done without invalidating the trace property. The trace property validity holds under the usual assumptions of perfect cryptography and Dolev-Yao attacker capabilities (e.g. no possibility for the attacker to guess secrets, or to decrypt encrypted messages without the proper key). As discussed in Section 3.4.2, this requires that the implementation parameters are chosen accordingly.

Another implicit assumption is that the Java abstract formal semantics, as defined in this paper, is correct, i.e. it accurately describes the actual behavior of the code run in a real Java virtual machine. Looking at Tables 6 and 8, this assumption mainly corresponds to assuming that the implementation of the cryptographic and communication operations is correct, i.e. coherent with the models used by ProVerif for these operations. For example, it is necessary to assume that $id_0.send(id_1)$ will really send the message stored in variable id_1 onto the channel whose reference is stored in variable id_0, without other side effects that may affect the future behavior. These last assumptions, in turn, are true if the SpiWrapper library and the marshaling layers used in the protocol are assumed to be correct.

For what concerns the marshaling layer, however, the correctness assumption is not strictly necessary; some weaker assumptions are enough, as detailed and proved in [PS14]. The reader is addressed to [PS14] in order to get a complete account of these weaker necessary assumptions. This also explains the choice of allowing the users of JavaSPI to use hand-written marshaling layers.

Finally, the soundness result has been obtained by introducing potential over-approximations in the applied π-calculus model (the possibility for all cryptographic operations to arbitrarily fail), which may lead to false positives reported by ProVerif. However, ProVerif itself introduces over-approximations, and the soundness result guarantees that, if ProVerif reports a property as satisfied in the abstract model, the same property is also satisfied when the concrete implementation of an actor substitutes its abstract model.

10. The SSL Case Study

The case study presented in [APPS11] has been extended in order to further validate the proposed JavaSPI approach. The previous version of the case study consisted of a simplified interoperable implementation of the SSL handshake protocol (both client and server) that implemented only one cipher suite. The example presented in this section covers a range of 22 cipher suites, which has been possible by extending the

marshaling layer and by exploiting the flexible annotation system presented in this paper, which crucially allows dynamic negotiation of cryptographic parameters.

The results of this case study show that JavaSPI can be effectively used to model and generate implementations of protocols that can be used in real-world distributed applications. The complete example is available online as part of the JavaSPI distribution (http://spi2java.polito.it/Downloads/JavaSPI_1.0.0.tar.gz).

The software developed in this case study can properly communicate with well known SSL implementations, such as OpenSSL, even if its functionality is limited to a single SSL 3.0 full handshake, and it does not support later versions of the protocol nor advanced protocol features such as: abbreviated handshakes; renegotiation; and client authentication.

Two different key exchange algorithms are modeled and supported in the case study: ephemeral Diffie-Hellman (with DSS and RSA signatures), and RSA key exchange. Both key exchange algorithms can be used in combination with different encryption algorithms. By using the "Bouncy Castle" library [The] as the Java provider of cryptographic operations, our SSL implementation supports many different encryption algorithms, including DES, Triple-DES, AES-128, AES-256, CAMELLIA-128, CAMELLIA-256, SEED and IDEA. This confirms the flexibility of JavaSPI, which can be integrated with existing cryptographic libraries.

Message Authentication Codes, used in SSL, are not present explicitly in the JavaSPI libraries but they can be modeled as keyed hashes. A keyed hash symbolically behaves like a hash function applied to the key-message pair. The SSL handshake does not use a standard MAC but it uses an ad-hoc algorithm to build MACs. In order to enable the development of an interoperable SSL implementation, we had to leverage the extendibility of JavaSPI by adding a specific data type for the SSL MAC. This has been achieved by extending the Hashing SpiWrapperSim class. According to this abstraction, the SSL MAC algorithm can be modeled as a standard hash during symbolic execution and formal verification, while an annotation specifies that the concrete implementation will use the proper MAC algorithm, hence achieving interoperability. It is important to note that this kind of extension regards the library, which is assumed to be correct. For this reason, this kind of extension is not considered as part of the normal usage of the framework, rather it is a maintenance task that should be done by the providers of the library only, after careful verification.

Following the JavaSPI development workflow, an abstract Java model has been developed for both the client and the server of the SSL3 handshake protocol, by using the SpiWrapperSim library. This enabled early prototyping of the protocol handshake, by means of the Java debugger. Specifically, we were able to check that message flow and content was as expected. Additionally, the Java compiler has been leveraged to check the static semantics correctness of the abstract model as Java code, while the Java-ProVerif and Java-Java generators have been used to check the model code satisfies the restrictions of the extended language.

The JavaSPI model consists of three different classes which extend *spiProcess*: `Client` and `Server` specify the behavior of the SSL client and server respectively, while `Master` orchestrates client and server instances. The `Master` process has also been used to simulate the protocol execution and it contains most of the annotations regarding the security properties to be checked during the formal verification phase.

The flows of client and server processes have two main branches: one to handle the message flow entailed by the ephemeral Diffie-Hellman key exchange, and another to handle the message flow of the RSA key exchange. Other parameters, such as the encryption algorithm and the signature type (DSS/RSA), are declared as parameters, resolved dynamically at runtime by means of annotations.

The following security properties have been expressed on the model:

- Secrecy, in client and server, of DH secret values (for the ephemeral Diffie-Hellman key exchange branches), and of the shared secret key (for the RSA key exchange branches).
- Secrecy of the private key (for the RSA key exchange branch).
- Server authentication to the client, expressed as an injective correspondence between the correct termination of the two processes: each time a client correctly terminates a session, agreeing on all relevant session data and the server identity, a server must have freshly started one session, agreeing on the same session data and the server identity. This applies both for ephemeral Diffie-Hellman and the RSA key exchange: two different injective correspondence properties have been specified, one for each key exchange algorithm.

Figure 9 shows the annotations used in the model to verify its security properties. For brevity, only the Master and the Client classes are shown. The Server mirrors the Client annotations, with minimal differences. Three types of annotations have been used:

- `@PVarDef`, used to define the initial attacker knowledge. At line 2, it is stated that every variable

```
1  public class Master extends spiProcess {
2    @PVarDef(PUBLIC)
3    @PQueryList({
4      @PEvinj({ "c_terminj_dhe", "s_beginj_dhe" }),
5      @PEvinj({ "c_terminj_rsa", "s_beginj_rsa" })
6    })
7    public void doRun() throws SpiWrapperSimException {
8      ...
9      final @PVarDef(PRIVATE) KeyPair p = new KeyPair();
10     final Certificate s_cert = (Certificate)p.getPublicKey();
11     final @PVarDef(PRIVATE) @PSecret PrivateKey s_prikey = p.getPrivateKey();
12     ...
13     //select the key exchange mode
14     //final Identifier DH_MODE = new Identifier("DHE");
15     final Identifier DH_MODE = new Identifier("NO-DHE");

16     final Server s = new Server( s_cert, s_prikey, ... );
17     final Clinet c1 = new Client ( s_cert, ... );
18     c1.send(DH_MODE);
19     spiProcess.start(s, c1);
20   }
21 }
22 ...
23 public class Client extends spiProcess {
24   public void doRun( ... ) throws SpiWrapperSimException {
25     ...
26     if(dhe.equals(DHE)){
27       // use DHE key exchange
28       ...
29       final @PSecret Integer DH_y = new Integer();
30       final A_Triplet c_DH_y_pl = new A_Triplet(DH_Q,   DH_y, DH_P);
31       final DHHashing c_DH_y = new DHHashing(c_DH_y_pl);
32       c.send(c_DH_y);
33       ...
34       final A_Triplet sharedPL = new A_Triplet(s_DH_x, DH_y, DH_P);
35       final DHHashing seedPL = new DHHashing(sharedPL);
36       final A_Triplet YXP_cs1 = new A_Triplet(seedPL,   c_rand, s_rand);
37       final Hashing small_key_C = new Hashing(YXP_cs1);
38       final A_Triplet YXP_cs2 = new A_Triplet(small_key_C, c_rand, s_rand);
39       final @PSecret Hashing key_C = new Hashing(YXP_cs2);
40       ...
41       if (hkMd5Sha.equals(myHkMd5Sha)) {
42         event("c_terminj_dhe", myHMD5,myHSHA, s_cert);
43       } else
44         fail();
45     } else {
46       // don't use DHE key exchange
47       ...
48       final @PSecret Pair<Message, Nonce> pre_master_secret = new Pair<Message, Nonce>(
             SSL_VERSION_3_0,nonce46);
49       ...
50       final A_Triplet YXP_cs1 = new A_Triplet(pre_master_secret, c_rand, s_rand);
51       final Hashing master_secret = new Hashing(YXP_cs1);
52       final A_Triplet YXP_cs2 = new A_Triplet(master_secret, c_rand, s_rand);
53       final @PSecret Hashing key_C = new Hashing(YXP_cs2);
54       ...
55       if (hkMd5Sha.equals(myHkMd5Sha)) {
56         event("c_terminj_rsa", myHMD5,myHSHA, s_cert);
57       } else
58         fail();
59     }
60   }
61 }
```

Fig. 9. An excerpt of the JavaSPI model containing annotations expressing the intended security properties

```
1  ...
2  (* Query about desired properties*)
3  query attacker:p.
4  query attacker:PriPart(p).
5  (* server process shared secret key - DHE *)
6  query attacker:H(((H(((DHKey(DH_x_11, DHPub(DH_y)),c_rand_1),s_rand_1)),c_rand_1),s_rand_1)).
7  (* server process PMS - DHE *)
8  query attacker:DHKey(DH_x_11, DHPub(DH_y)).
9  (* server process secret DHx - DHE*)
10 query attacker:DH_x_11.
11 (* client process shared secret key -DHE *)
12 query attacker:H(((H(((DHKey(DH_y, DHPub(DH_x_11)),c_rand),s_rand)),c_rand),s_rand)).
13 (* client process PMS - DHE *)
14 query attacker:DHKey(DH_y, DHPub(DH_x_11)).
15 (* client process secret DHy - DHE *)
16 query attacker:DH_y.
17 (* client process PMS - RSA *)
18 query attacker:(SSL_VERSION_3_0,nonce46).
19 (* client process shared secret key - RSA *)
20 query attacker:H(((H((((SSL_VERSION_3_0,nonce46),c_rand),s_rand)),c_rand),s_rand)).

21 query ev:evt_s_beginj_dhe(x,xx,xxx).
22 query ev:evt_s_term_dhe().
23 query ev:evt_s_term_rsa().
24 query ev:evt_s_beginj_rsa(x,xx,xxx).
25 query ev:evt_c_terminj_rsa(x,xx,xxx).
26 query ev:evt_c_terminj_dhe(x,xx,xxx).
27 query evinj:evt_c_terminj_rsa(x,xx,xxx) ==> evinj:evt_s_beginj_rsa(x,xx,xxx).
28 query evinj:evt_c_terminj_dhe(x,xx,xxx) ==> evinj:evt_s_beginj_dhe(x,xx,xxx).
29 ...
30 out(c1,DH_MODE);
31 (
32  (!
33   in(c1,dhMode);
34   ...
35   0
36  )
37  |
38  (!
39   in(c,c_hello_1);
40   ...
41   0
42  )
43 )
```

Fig. 10. An excerpt of the generated ProVerif model, showing the security queries generated by the annotations in Figure 9

declared inside the doRun method belongs to the initial attacker knowledge. This method contains all the configuration parameters that the client and the server share at startup, for which there is no expectation of secrecy, hence the `PUBLIC` property. The only exception is the key pair and its private key, annotated as private at lines 9 and 11: the private key is private because, by definition, only the Server must know it, while the key pair is also private because, even if it contains a public part known to the attacker (the certificate), it also contains the private key, hence it could not be considered as part of the attacker knowledge.

- `@PSecret`, used to assert secrecy claims about certain variables, that the model checker will then have to prove. In the RSA-based SSL handshake, secrecy of the certificate private key (at line 11), the pre-master secret and the encryption keys (at lines 48-53) is claimed. In the DH key exchange, annotations at lines 29 and 39 claim that the random nonces generated by client and server and the final key generated after the exchange are secret.
- `@PEvInj`, used to declare the authentication properties. These annotations work in conjunction with the

```
1  @HashingA(ConcreteClass = "FinalHashNoEncode", Algo = "SHA-1", Provider = "SUN")
2  @SharedKeyCipheredA(ConcreteClass = "FinishedRecv", Padding = "NoPadding", Provider = "SunJCE;BC"
       )
3  @SharedKeyA(ConcreteClass="SKSSL")
4  public class Client extends spiProcess {

5   public void doRun(...){
6    ...
7    final Identifier encryption = s_ciphSuite.getEncryption();
8    final Identifier dhe = s_ciphSuite.getDhe();
9    ...
10   if(dhe.equals(DHE)){
11    //ephemeral Diffie-Hellman cipher suites
12    ...
13    @ConcreteClass("RawBytesIV")
14    final Hashing s_write_iv = new Hashing(PB3);
15    ...
16    @Iv(value="((SSLIVExtractor)s_write_iv).getText()",type=Types.varName)
17    @Algo(value="encryption",type=Types.varName)
18    final SharedKeyCiphered<A_FinishFullMessage> fin =
19         c.receive(SharedKeyCiphered.class);
20    ...
21   }else{
22    //RSA key exchange cipher suites
23    ...
24   }
25  }
26 }
```

Fig. 11. A portion of the annotated SSL client JavaSPI model

named `event` methods, called at both the beginning of each server handshake branch and at the end of each client handshake branch (lines 42, 56).

The annotated model is used to generate the ProVerif model, which is then passed to ProVerif for automatic verification of the secrecy and authentication properties. Figure 10 shows the generated ProVerif security property queries, previously defined in terms of annotations in Figure 9. There is a direct correspondence between the previously defined annotations and each set of ProVerif queries.

According to the JavaSPI development workflow, once the ProVerif model has been verified, an SSL-specific marshaling library can be developed, and refinement annotations added to the JavaSPI model, so that interoperable code can be generated.

Figure 11 shows an excerpt of code containing annotations from the `Client` model. Static details of each model object type, specified in the annotation applied to the class declaration (lines 1-3), are extended as default values to all objects belonging to that type. However, these values may be not suitable for all objects. For example, annotation on line 13 overrides the value `ConcreteClass` only for the `s_write_iv` variable. Dynamic values in annotations are necessary when a value is known only at runtime. For example, the encryption algorithm is selected by the server (in the non-trivial case where more cipher suites are supported), and the initialization vectors are extracted from the master secret at runtime. In Figure 11, line 16 specifies that the initialization vector of the ciphered object `fin` is the runtime value of variable `s_write_iv` (after being casted to `SSLIVExtractor`). In the same way, the encryption algorithm identifier is read by the client and saved into `encryption` variable (line 7). Finally, the algorithm is specified as parameter of the ciphered message (line 17).

The marshaling library is composed of several classes like the one presented in Figure 12. Each one of them defines a method to transform itself into a binary data stream, and another method to parse a binary data stream and rebuild the same object. According to the sufficient soundness conditions expressed in [PS14], the implementation of these classes has been kept minimal, only performing network serialization checks whose failure is equivalent to the attacker tampering with the communication channel.

The complete JavaSPI model (client and server) of this case study required a total of about 180 annotations, which amounts to about 13% of the whole model size.

The generated client and server implementations have been successfully tested for interoperability against

```
1  package it.polito.ssl.marsh;

2  public class IDSHelloDone extends Identifier {

3    private final byte[] header;
4    private final byte[] payload;

5          ...
6          public IDSHelloDone(String text) {
7                  super(text);
8                  header = ...
9                  payload = ...
10         }
11         ...
12         @Override
13         protected void _serialize(OutputStream out) throws PostException {
14                 try {
15                         out.write(header);
16                         out.write(payload);
17                         out.flush();
18                 } catch (IOException e) {
19                         throw new PostException(e);
20                 }
21         }
22         @Override
23         public Message deSerialize(InputStream in) throws PostException {
24                 DataInputStream din = new DataInputStream(in);

25                 try {
26                         byte[] header = new byte[5];
27                         din.readFully(header);
28                         ...
29                         int type = din.read();
30                         if (type!=HandshakeTypes.SERVERHELLODONE.getValue()) {
31                                 throw new PacketException("Wrong␣Handshake␣message␣type:␣found␣"+
                                       type+";␣expected␣"+HandshakeTypes.SERVERHELLODONE.getValue()+
                                       ".");
32                         }
33                         ...
34                         return this;
35                 } catch (IOException e) {
36                         throw new PostException(e);
37                 }
38         }
39 }
```

Fig. 12. A sample object in the marshaling layer

OpenSSL 1.0.1h. As our automatically generated implementations have some limitations with respect to the OpenSSL client and server, during the interoperability tests the OpenSSL client and server have been run with proper parameters, in order to avoid the abort of the handshake. More precisely, client and server have been run with the -ssl3 option, which avoids the use of SSL versions other than 3, and the server has been run with the default option of not using client authentication and by enabling only the cipher suites we have implemented.

From the security point of view, the implementations of SSL generated in this way are guaranteed to satisfy the secrecy and authentication properties that have been verified on the ProVerif model. This means that they are free from attacks that could be discovered with a Dolev-Yao attacker model and perfect cryptography assumption. This holds under the implicit assumptions that the SpiWrapper library and the JavaSPI semantics are correct. Of course, different kinds of attacks, e.g. covert channels, cannot be prevented by simply using JavaSPI. They need other complementary measures.

In traditional development, debugging a security protocol implementation is a complex task, because little information is intentionally leaked by cryptographic primitives when an error occurs. As a consequence, developing a new interoperable implementation requires significant development effort, and is error prone,

because issues related to the marshaling layer could easily interfere with the correct development of the core protocol logic. Thanks to the JavaSPI workflow, it has been possible to completely decouple the protocol model development from the development of its concrete implementation: at first, in fact, the SpiWrapperSim library allowed us to debug and verify the correctness of symbolic protocol models, before even starting to design the marshaling layer. Then, only after having completed and proved the security properties of the model, the task of annotating the model and developing its interoperability layer was undertaken.

11. Conclusion

This paper presented JavaSPI, a framework for modeling, verification and implementation of cryptographic protocols following a model-driven approach. JavaSPI facilitates proper implementation of cryptographic protocols by developers not familiar with formal languages and formal verification techniques.

The main features that JavaSPI combines together are: i) protocol modeling stage guided through the use of the SpiWrapperSim library, ii) the possibility to simulate the execution of the protocol and analyze the steps with the Java debugger, iii) automatic verification of security properties using the tool ProVerif, iv) generation of Java code that implements the protocol, including all the details specified by annotations in the abstract model. A formalization of the Java modeling language, of its translation into the ProVerif language, and its refinement into an implementation have been provided along with a proof of the soundness of the refinement process.

References

[ABB+10] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on Σ-protocols. In *15th European Conference on Research in Computer Security (ESORICS 2010)*, volume 6345 of *Lecture Notes in Computer Science*, pages 151–167, Berlin, Heidelberg, 2010. Springer-Verlag.

[ABBD13] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *2013 ACM SIGSAC Conference on Computer & Communications Security (CCS 2013)*, pages 1217–1230. ACM, 2013.

[ACMR12] Tigran Avanesov, Yannick Chevalier, Mohammed Anis Mekki, and Michaël Rusinowitch. Web services verification and prudent implementation. In *Data Privacy Management and Autonomous Spontaneus Security*, volume 7122 of *Lecture Notes in Computer Science*, pages 173–189. Springer-Verlag, Berlin, Heidelberg, 2012.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *ACM SIGPLAN Notices*, 36(3):104–115, 2001.

[AFP13] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy (S&P 2013)*, pages 526–540. IEEE, 2013.

[AGJ12] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational verification of C protocol implementations by symbolic execution. In *2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 712–723. ACM, 2012.

[App14] Apple goto fail bug, 2014. CVE-2014-1266, `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266`.

[APPS11] Matteo Avalle, Alfredo Pironti, Davide Pozza, and Riccardo Sisto. JavaSPI: A framework for security protocol implementation. *International Journal of Secure Software Engineering*, 2(4):34–48, 2011.

[APS14] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, 2014.

[APSP11] Matteo Avalle, Alfredo Pironti, Riccardo Sisto, and Davide Pozza. The JavaSPI framework for security protocol implementation. In *6th International Conference on Availability, Reliability and Security (ARES 2011)*, pages 746–751. IEEE Computer Society, 2011.

[APW09] Martin R. Albrecht, Kenneth G. Paterson, and G. Watson. Plaintext recovery attacks against ssh. In *IEEE Symposium on Security and Privacy (S&P 2009)*, pages 16–26. IEEE, 2009.

[BCD+09] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 124–140. IEEE Computer Society, 2009.

[BCPP+12] Piergiuseppe Bettassa Copet, Alfredo Pironti, Davide Pozza, Riccardo Sisto, and Pietro Vivoli. Visual model-driven design, verification and implementation of security protocols. In *IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE 2012)*, pages 62–65. IEEE, 2012.

[BDL06] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[BFGT08] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems*, 31(1):1–61, 2008.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE workshop on Computer Security Foundations (CSF 2001)*, pages 82–96. IEEE Computer Society, 2001.

[Bla09] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[BLF+14] Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy (S&P 2014)*, pages 98–113. IEEE, 2014.

[BPP03] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, 2003.

[CB12] David Cade and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *7th International Conference on Availability, Reliability and Security (ARES 2012)*, pages 65–74. IEEE Computer Society, 2012.

[DGJN14] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. *Journal of Computer Security*, 22(5):823–866, 2014.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[Gnu14] GnuTLS certificate verification issue, 2014. CVE-2014-0092, `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0092`.

[GV15] Roberto Gorrieri and Cristian Versari. *Transition Systems and Behavioral Equivalences*, pages 21–79. Springer International Publishing, 2015.

[Hea14] Heartbleed bug, 2014. CVE-2014-0160, `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160`.

[HT96] Nevin Heintze and J.D. Tygar. A model for secure protocols and their compositions. *IEEE Transactions on Software Engineering*, 22(1):16–30, 1996.

[Jür01] J. Jürjens. Secrecy-preserving refinement. In J. Fiadeiro and P. Zave, editors, *International Symposium on Formal Methods Europe (FME)*, volume 2021 of *Lecture Notes in Computer Science*, pages 135–152, Berlin Heidelberg New York, 2001. Springer Verlag.

[Jür05] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[KOT08] Shinsaku Kiyomoto, Haruki Ota, and Toshiaki Tanaka. A security protocol compiler generating C source codes. In *Information Security and Assurance*, pages 20–25. IEEE, 2008.

[MJH+10] Lionel Montrieux, Jan Jürjens, Charles B. Haley, Yijun Yu, Pierre-Yves Schobbens, and Hubert Toussaint. Tool support for code generation from a umlsec property. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pages 357–358. ACM, 2010.

[O'S08] Nicholas O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, pages 221–226, 2008.

[PS07] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *IEEE Symposium on Computers and Communications*, pages 839–844. IEEE, 2007.

[PS14] Alfredo Pironti and Riccardo Sisto. Safe abstractions of data encodings in formal security protocol models. *Formal Aspects of Computing*, 26(1):125–167, 2014.

[RRDO10] Eric Rescorla, Marsh Ray, Steve Dispensa, and Nasko Oskov. Transport layer security (TLS) renegotiation indication extension, 2010. RFC 5746.

[SPP01] Dawn Xiaodong Song, Adrian Perrig, and Doantam Phan. AGVI - automatic generation, verification, and implementation of security protocols. In *13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 241–245, Berlin, Heidelberg, 2001. Springer-Verlag.

[The] The Legion of Bouncy Castle. Bouncy Castle crypto API. `http://www.bouncycastle.org/java.html`.